Synchronization

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.

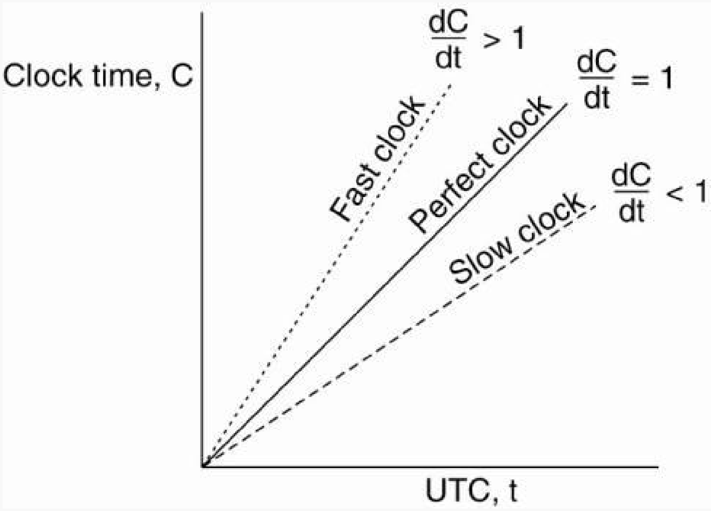
UTC is **broadcast** through short wave radio and satellite.

- Satellites can give an accuracy of about ±0.5 ms.

Problem: Suppose we have a distributed system with a UTC-receiver somewhere in it => 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, $dC/dt = 1$.



In practice:

$$1-\rho=\frac{dC}{dt}\leq 1+\rho$$

Where *r i*s the maximum drift rate

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

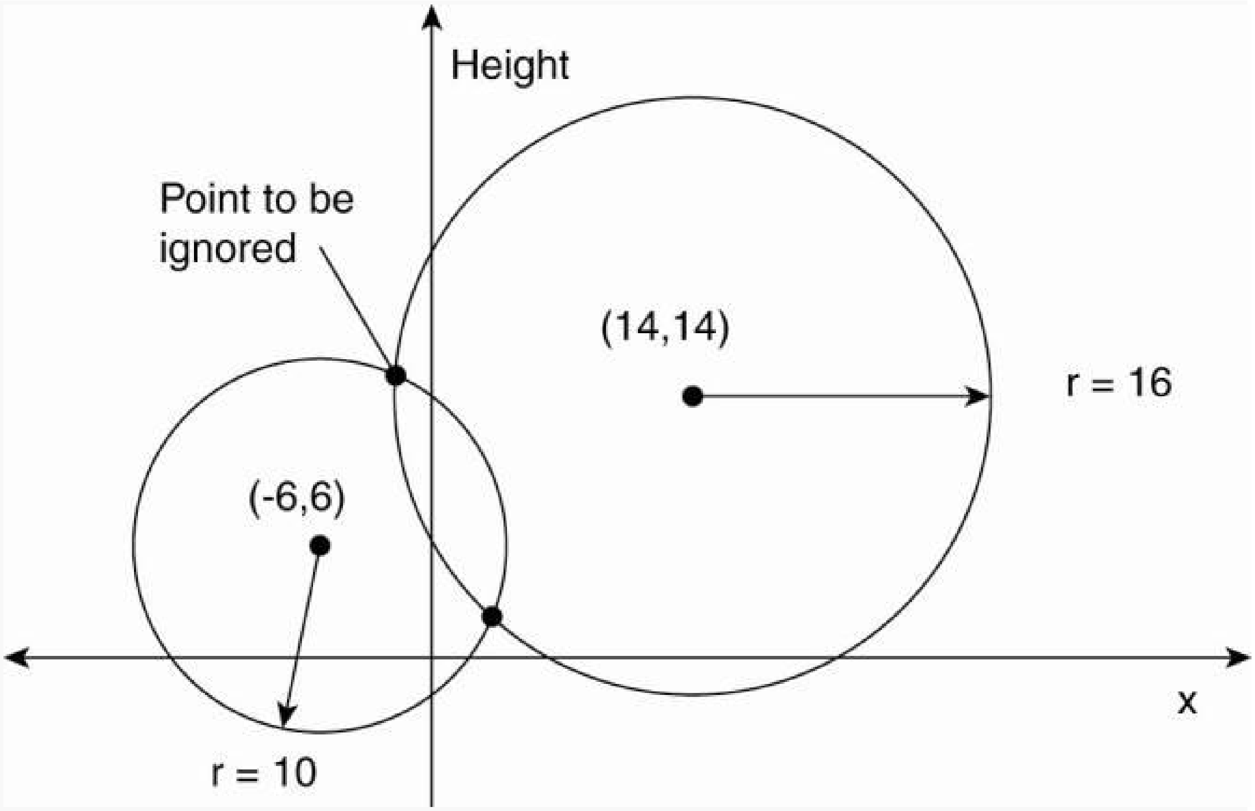
Services:
[NTP (Network Time Protocol)]() and Servers
[National Institute of Standard Time (NIST)]()
[http://www.worldtimeserver.com/]()

Global Positioning System (GPS)
GPS verview by Zogg ([2002]())

- GPS uses 29 satellites each circulating in an orbit at a height of approximately 20,000 km.
- Each satellite has up to four atomic clocks, which are regularly calibrated from special stations on Earth.
- A satellite continuously broadcasts its position, and time stamps each message with its local time.
- This broadcasting allows every receiver on Earth to accurately compute its own position using, in principle, only three satellites.

Basic idea: You can get an accurate account of the time as a side-effect of GPS.

Principle:



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

- *t* takes a while before a signal reaches the receiver
- The receiver's clock is definitely out of synch with the satellite

- Δ_r is unknown deviation of the receiver's clock.

- x_r, y_r, z_r are unknown coordinates of the receiver.

- T_i is timestamp on a message from satellite i

- $\Delta_i = (T_{now} - T_i) + \Delta_r$ is 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

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

4 satellites Þ 4 equations in 4 unknowns (with Δ_r as one of them):

$$d_i + c\Delta_r = c\,\Delta_i$$

Clock Synchronization Algorithms

A survey is given in Ramanathan et al. (1990).

Network Time Protocol (NTP)

NTP is a protocol designed to synchronize the clocks of computers over a network.

Clock Synchronization Principles

Principle I: Every machine asks a **time server (**Cristian 1989) for the accurate time at least once every $\delta / (2\,\rho)$ seconds (**Network Time Protocol**).

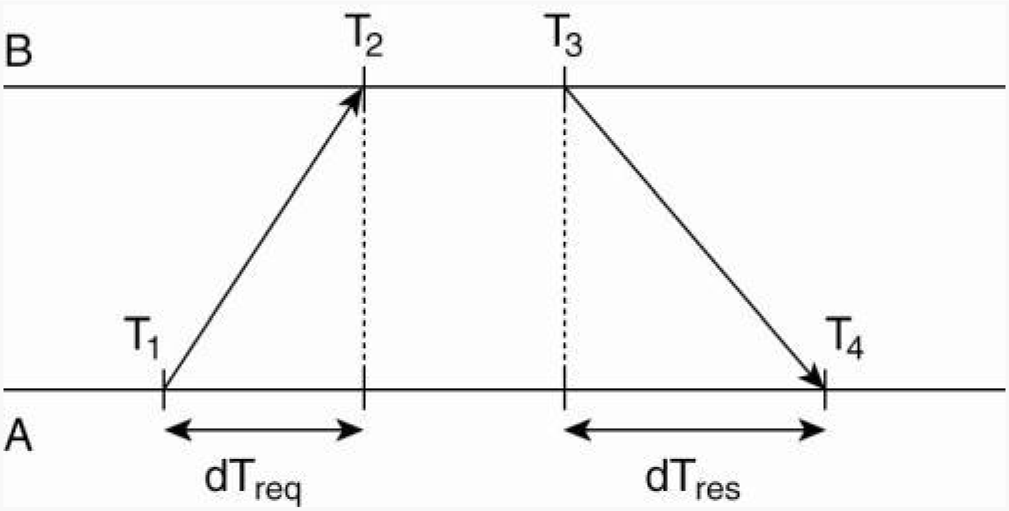
- Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.

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.

- Okay, you'll probably get every machine in sync. **Note:** 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 Þ smooth adjustments.

Getting the current time from a time server.



1. A will send a request to B, timestamped with value T1 .
2. B will record the time of receipt T2 (taken from its own local clock), and return a response timestamped with value T3, piggybacking the previously recorded value T2.
3. A records the time of the response's arrival, T4.

- Assume that the propagation delay from A to B is the same as B to A, meaning that T2 - T1 ≈ T4 -T3.
- A can estimate its offset relative to B as

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

- If A's clock is fast => $\theta < 0$, A cannot set its clock backward. Not allowed
- Must slow down clock gradually.
    - e.g. timer is set to generate 100 interrupts per second.
        - each interrupt would add 10 msec to the time.
        - When slowing down, the interrupt routine adds only 9 msec each time until the correction has been made.
        - Vice versa for advancing

- In the case of the network time protocol (NTP), this protocol is set up pair-wise between servers. In other words, B will also probe A for its current time.
- The offset θ is computed, along with the estimation δ for the delay:

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

- Eight pairs of (θ,δ) values are buffered, finally taking the minimal value found for δ as the best estimation for the delay between the two servers, and subsequently the associated value θ as the most reliable estimation of the offset.

Issue: clock accuracy

- Some cocks are more accurate than
- NTP divides servers into strata.
- When A contacts B, it will only adjust its time if its own stratum level is higher than that of B.
    - After synchronization, A's stratum level will become one higher than that of B.
    - Due to the symmetry of NTP, if A's stratum level was lower than that of B, B will adjust itself to A.

Stratum 0

Devices such as atomic (cæsium, rubidium) clocks, GPS clocks or other radio clocks.
Stratum-0 devices are not attached to the network; instead they are locally connected to computers (e.g. via an RS-232 connection using a Pulse per second signal).

Stratum 1

Computers attached to Stratum 0 devices.
Normally they act as servers for timing requests from Stratum 2 servers via NTP.
Also referred to as timeservers.

Stratum 2

Computers that send NTP requests to Stratum 1 servers.
Normally a Stratum 2 computer will reference a number of Stratum 1 servers and use the NTP algorithm to gather the best data sample, dropping any Stratum 1 servers that seem obviously wrong.
Stratum 2 computers act as servers for Stratum 3 NTP requests.

Stratum 3 and higher

These computers employ exactly the same NTP functions of peering and data sampling as Stratum 2, and can themselves act as servers for higher strata, potentially up to 16 levels. NTP (depending on what version of NTP protocol in use) supports up to 256 strata.

- Reviews of NTP implementation:

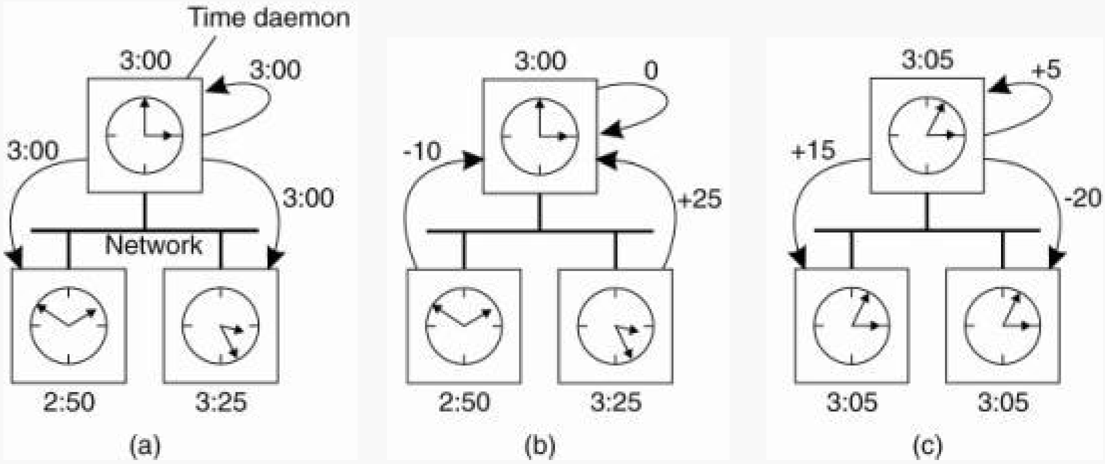
Mills (1992) and (2006).

The Berkeley Algorithm

- Many algorithms such as NTP, the time server is passive. - Other machines periodically ask it for the time.
- Opposite - Berkeley UNIX (Gusella and Zatti, 1989) time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there.

- Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved.
- This method is suitable for a system in which no machine has a WWV receiver.
- The time daemon's time must be set manually by the operator periodically.

(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.



1. at 3:00, the time daemon tells the other machines its time and asks for theirs.
2. they respond with how far ahead or behind the time daemon they are.
3. the time daemon computes the average and tells each machine how to adjust its clock [

Note: many purposes - sufficient that all machines agree on the same time.

Logical Clocks

The Happened-Before Relationship

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

The **happened-before** relation on the set of events in a distributed system:

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

Note: this introduces a **partial ordering of events** in a system with concurrently operating processes.

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 $C(a) < C(b)$.

Lamport's Logical Clocks Lamport (1978)

Problem: How to attach a timestamp to an event when there's no global clock => maintain a **consistent** set of logical clocks, one per process.

Solution: Each process P_i maintains a **local** counter C_i and adjusts this counter according to the following rules: (Raynal and Singhal, 1996)

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 $ts(m) = C_i$.

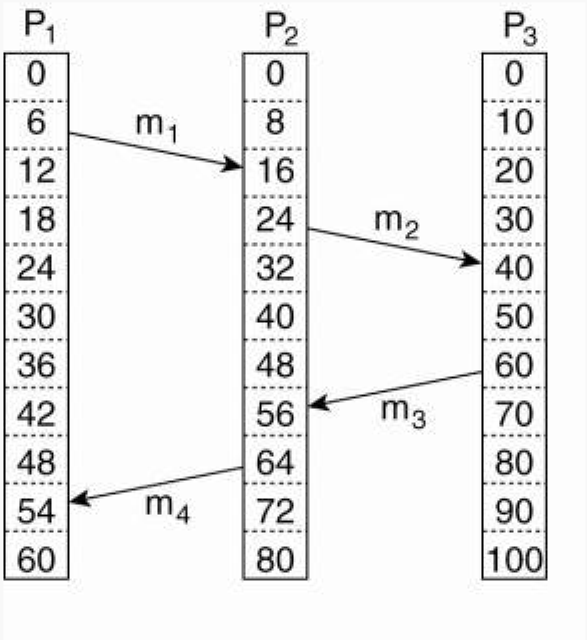
3: Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

Property **P1** is satisfied by (1);
Property **P2** by (2) and (3).

Note: it can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

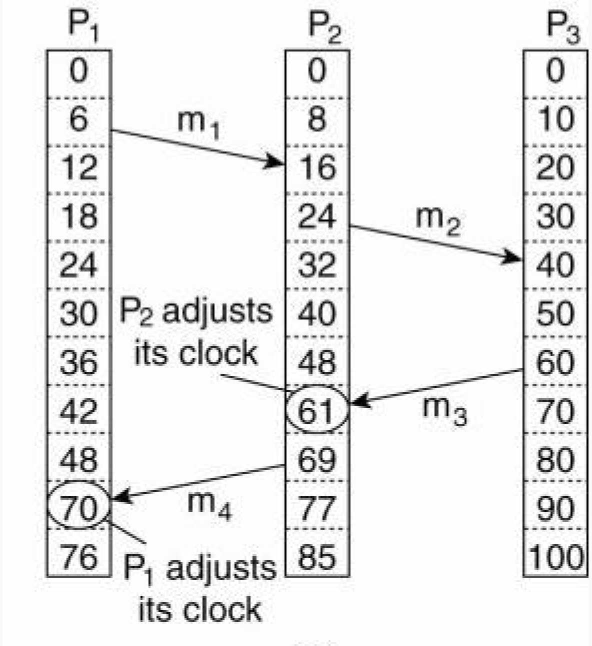
Example

(a) Three processes, each with its own clock. The clocks run at different rates.
(b) Lamport's algorithm corrects the clocks.



P1	P2	P3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

(a)



P1	P2	P3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

(b)

- The processes run on different machines, each with its own clock, running at its own speed.
- When the clock has ticked 6 times in process P1, it has ticked 8 times in process P2 and 10 times in process P3.
- Each clock runs at a constant rate, but the rates are different due to differences in the crystals.

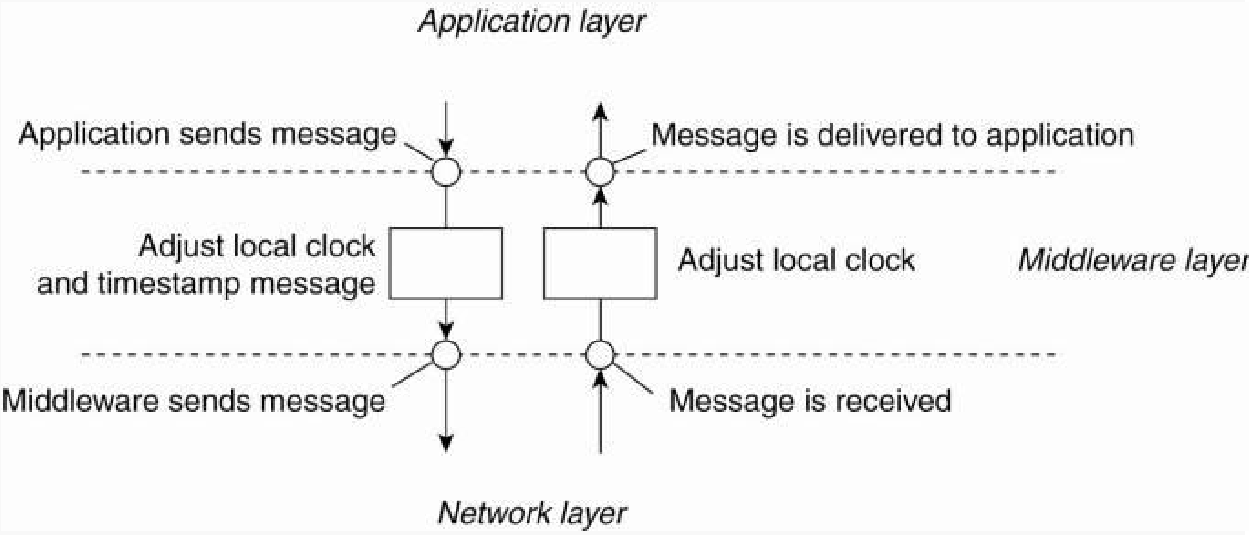
- At time 6, process P1 sends message m1 to process P2, the clock in process P2 reads 16 when it arrives.

- If the message carries the starting time, 6, in it, process P2 will conclude that it took 10 ticks to make the journey.

- m3 leaves process P3 at 60 and arrives at P2 at 56.
- m4 from P2 to P1 leaves at 64 and arrives at 54.
- These values are clearly impossible. It is this situation that must be prevented.

- Since m3 left at 60, it must arrive at 61 or later.
  - Each message carries the sending time according to the sender's clock.
  - When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time.

Note: Adjustments take place in the middleware layer:

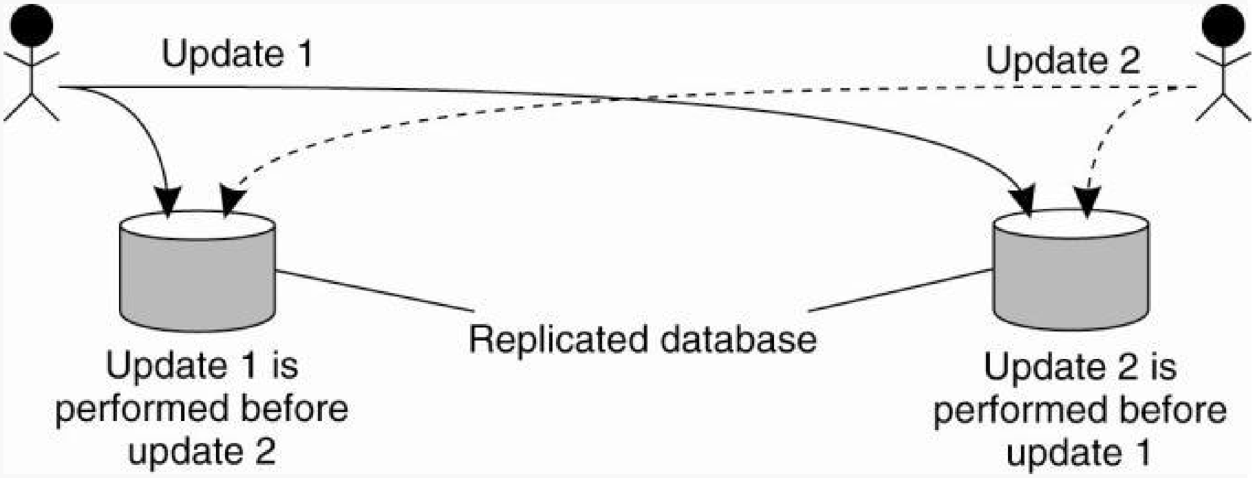


Example: Totally Ordered Multicast

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

Updating a replicated database and leaving it in an inconsistent state.


Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

Result: in absence of proper synchronization: replica #1 $1111, while replica #2 $1110.

Solution:

- Process *Pi* sends timestamped message *msgi* to all others. The message itself is put in a local queue *queuei*.
- Any incoming message at *Pj* is queued in *queuej*, according to its timestamp, and acknowledged to every other process.

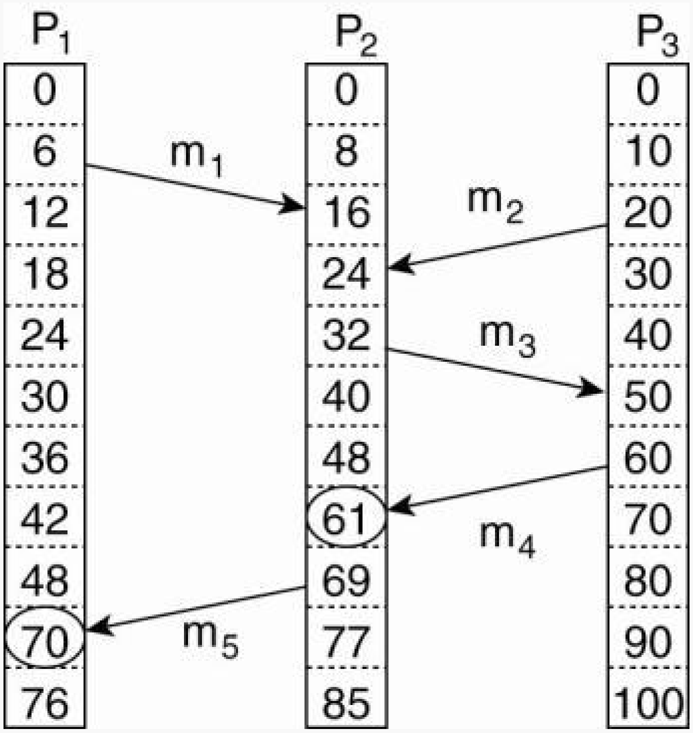
Pj passes a message *msgi* to its application if:
(1) *msgi* is at the head of *queuej*
(2) for each process *Pk*, there is a message *msgk* in *queuej* with a larger timestamp.

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.
- We **cannot** conclude that *a* causally precedes *b*.

Solution: vector clocks

- 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 receives a message *m* 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.

Enforcing Causal Communication

Causally Ordered Multicasting

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

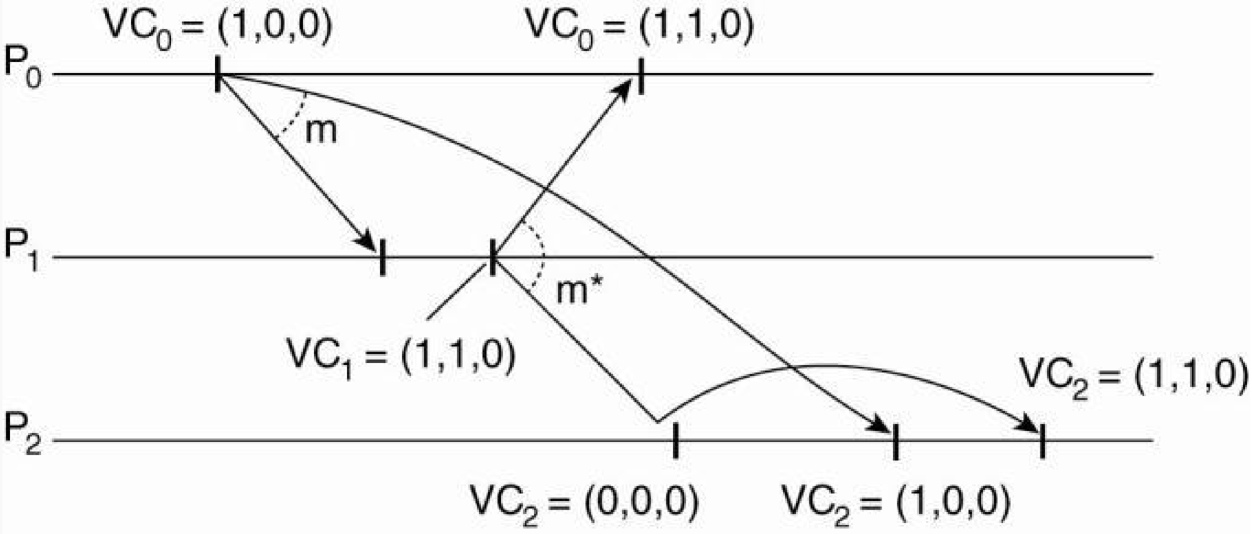
Adjustment: Pi increments $VC_i[i]$ only when sending a message, and P_j only adjusts VC_j when receiving a message (i.e., does not increment $VC_j[j]$.

Pj postpones delivery of *m* until:

- $ts(m)[i] = VC_j[i] + 1$.
    - o Means that m is the next message that P_j was expecting from process P_i
- $ts(m)[k] \leq VC_j[k]$ for $k \neq j$
    - o Means that P_j has seen all the messages that have been seen by P_i when it sent message m.
-

Example:

Enforcing causal communication.



- Three processes P0, P1, and P2.
  - At local time (1,0,0), P1 sends a message *m* to the other two processes.
  - After its receipt by P1, P1 sends *m**, which arrives at P2 sooner than *m*.
  - At that point, the delivery of *m** is delayed by P2 until *m* has been received and delivered to P2's application layer.

A Note on Ordered Message Delivery

Issue: middleware systems provide support for totally-ordered and causally-ordered (reliable) multicasting.

Problem:

- The middleware cannot tell what a message actually contains, only potential causality is captured.
- Not all causality may be captured.

Solution: end-to-end argument: (Saltzer et al., 1984)

Ordering issues can be solved by looking at the application for which communication is taking place.

Mutual Exclusion

Survey of distributed algorithms for mutual exclusion is provided by Velazquez (1993).

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**.

Centralized: Really simple:

Distributed mutual exclusion algorithms:

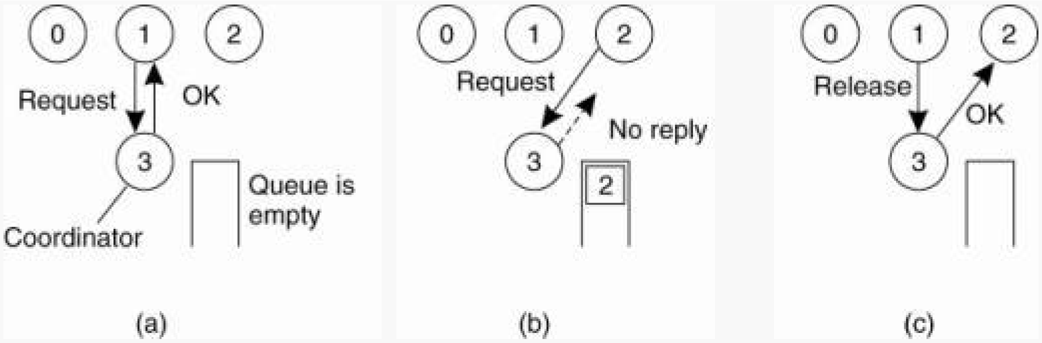
1. Token-based solutions: mutual exclusion is achieved by passing a special message between the processes, known as a token.
    - Only one token available and who ever has that token is allowed to access the shared resource.
    - When finished, the token is passed on to a next process.
    - Properties:
        1. Ensure that every process will get a chance at accessing the resource.
        2. Deadlocks are avoided.
        3. Drawback - token loss leads to a complex restart of procedure
2. Permission-based approach: a process wanting to access the resource first requires the permission of other processes.

Centralized Algorithm

Simulate how it is done in a one-processor system

- One process is elected as the coordinator.
- Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission.
- If no other process is currently accessing that resource, the coordinator sends back a reply granting permission
- When the reply arrives, the requesting process can proceed.

(a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.
(b) Process 2 then asks permission to access the same resource. The coordinator does not reply.
(c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.



Issues:

- The coordinator is a single point of failure, so if it crashes, the entire system may go down.
- In a large system, a single coordinator can become a performance bottleneck.

Decentralized Mutual Exclusion – (Lin et al. 2004)

A voting algorithm that can be executed using a DHT-based system

Principle: Assume every resource is replicated n times, with each replica having its own coordinator Þ 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.

Issue: How robust is this system? Let p denote the probability that a coordinator crashes and recovers in a period Dt Þ probability that k out m coordinators reset:

$$P[k] = \begin{bmatrix} m \\ k \end{bmatrix} p^k (1-p)^{m-k}$$

Given that at least 2m - n coordinators need to reset in order to violate the correctness of the voting mechanism, the probability that such a violation occurs is then $\sum_{k=2m-n}^{n} P[k]$.

Example:

- A DHT-based system in which each node participates for about 3 hours in a row.
- Let Δt be 10 seconds, which is considered to be a conservative value for a single process to want to access a shared resource.
- With n = 32 and m = 0.75n.
- The probability of violating correctness : $p_v < 10^{-40}$

A Distributed Algorithm

Mutual Exclusion: Ricart & Agrawala

Deterministic distributed mutual exclusion algorithm

- Lamport's 1978 paper on clock synchronization presented the first
- Ricart and Agrawala (1981) made it more efficient.

- Requires that there be a total ordering of all events in the system
  - for any pair of events - it must be unambiguous which one actually happened first.

- When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time.
- It then sends the message to all other processes, conceptually including itself.
- When a process receives a request message from another process, the action it takes depends on its own state with respect to the resource named in the message.
- Three different cases have to be clearly distinguished:

1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.

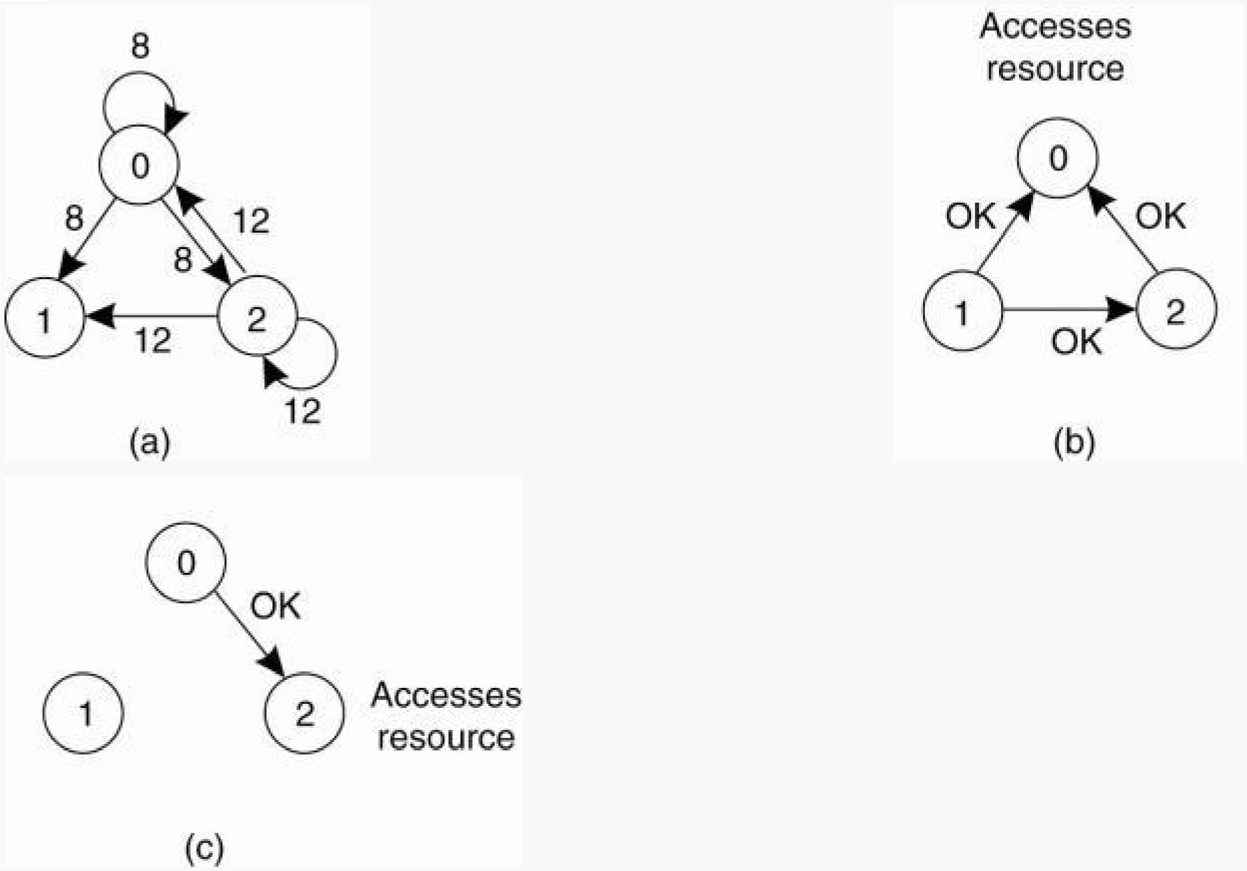
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.

3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone.
   - The lowest one wins.
     a. If the incoming message has a lower timestamp, the receiver sends back an OK message.
     b. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

- After sending out requests asking permission, a process waits until everyone else has given permission – then proceeds.
- When finished, it sends OK messages to all processes on its queue and deletes them all from the queue.

- If there is no conflict, it clearly works.
- Suppose that two processes try to simultaneously access the resource:

(a) Two processes want to access a shared resource at the same moment.
(b) Process 0 has the lowest timestamp, so it wins.
(c) When process 0 is done, it sends an OK also, so 2 can now go ahead.



1. Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12.
2. Process 1 is not interested in the resource, so it sends OK to both senders.
3. Processes 0 and 2 both see the conflict and compare timestamps.
4. Process 2 sees that it has lost, so it grants permission to 0 by sending OK. Process 0 now queues the request from 2 for later processing and access the resource, as shown in Fig. 6figure(b).
5. When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to go ahead, as shown in Figure(c).

· The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps.

Advantages:

· Mutual exclusion is guaranteed without deadlock or starvation.

· The number of messages required per entry is now 2(n - 1), where the total number of processes in the system is n.

- No single point of failure exists.

Problem 1:

- Single point of failure has been replaced by n points of failure.
  - § If any process crashes, it will fail to respond to requests.
  - § Silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions.
  - § Since the probability of one of the n processes failing is at least n times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is more than n times worse and requires much more network traffic as well.

Solution:

- When a request comes in, the receiver always sends a reply, either granting or denying permission.
- Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead.
- After a request is denied, the sender should block waiting for a subsequent OK message.

Problem 2:

- Either a multicast communication primitive must be used or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing.
  - o The method works best with small groups of processes that never change their group memberships.

Problem 3:

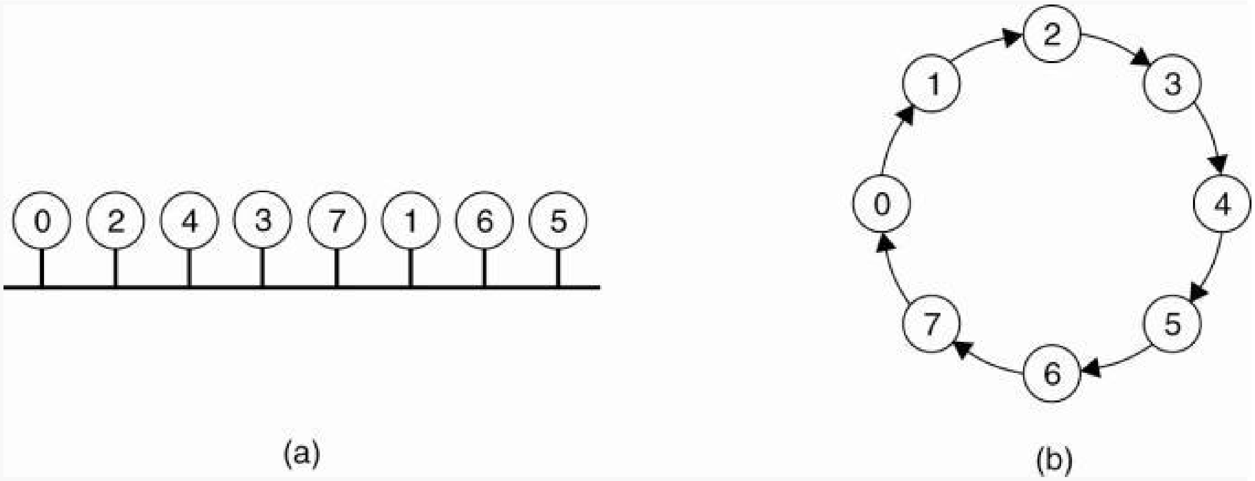
- A centralized algorithm that handles all requests can lead to a bottlenecks.
- In the distributed algorithm, all processes are involved in all decisions concerning accessing the shared resource.
  - o If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is not going to help much.

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) An unordered group of processes on a network. (b) A logical ring constructed in software.



(a)

(b)

- When the ring is initialized, process 0 is given a token.
- The token circulates around the ring.
- It is passed from process k to process k+1 (modulo the ring size) in point-to-point messages.
- When a process acquires the token from its neighbor, it checks to see if it needs to access the shared resource.
  - If so, the process goes ahead, does all the work it needs to, and releases the resources.
  - After it has finished, it passes the token along the ring.
  - It is not permitted to immediately enter the resource again using the same token.
- If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along.
  - As a consequence, when no processes need the resource, the token just circulates at high speed around the ring.

Advantages:

- Only one process has the token at any instant, so only one process can actually get to the resource.
- Since the token circulates among the processes in a well-defined order, starvation cannot occur.
- Once a process decides it wants to have access to the resource, at worst it will have to wait for every other process to use the resource.

Problems:

- If the token is ever lost, it must be regenerated
  - Dead processes:
    - If a process receiving the token must acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails.
    - At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary.

Comparison of the Four Algorithms

A comparison of three mutual exclusion algorithms.

Algorithm	*Messages per entry/exit*	*Delay before entry (in message times)*	*Problems*
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1, 2,...	2 m	Starvation, low efficiency
Distributed	2(n - 1)	2(n - 1)	Crash of any process
Token ring	1 to	0 to n - 1	Lost token, process crash

Global Positioning of Nodes

Geometric overlay networks

- Each node is given a position in an m-dimensional geometric space, such that the distance between two nodes in that space reflects a real-world performance metric.
- Example: distance corresponds to internode latency.
  - Given two nodes P and Q, then the distance $d(P,Q)$ reflects how long it would take for a message to travel from P to Q.and vice versa.

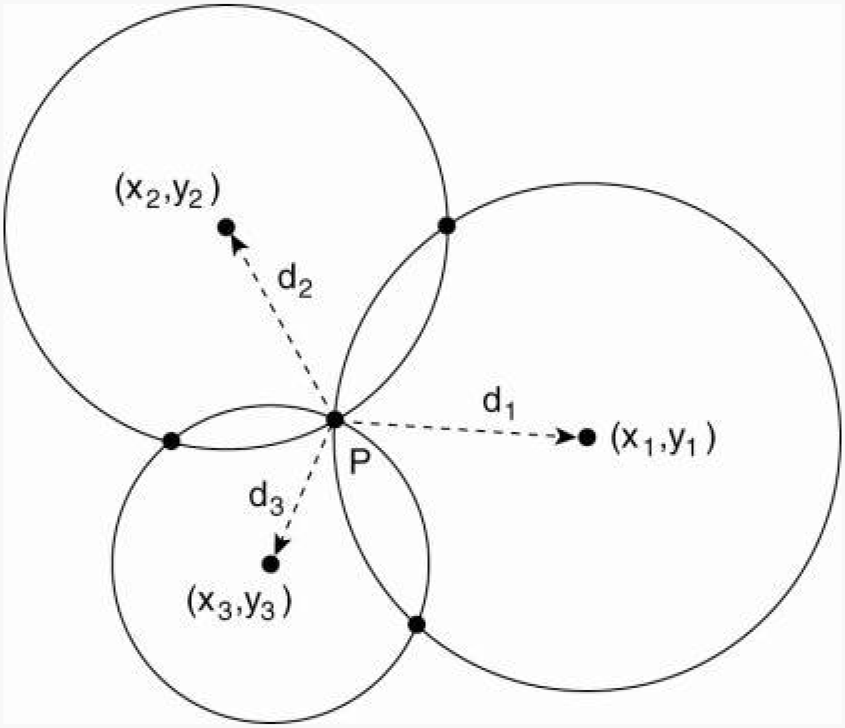
Problem: How can a single node efficiently estimate the latency between any two other nodes in a distributed system?

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

Observation: a node P needs $k + 1$ **landmarks** to compute its own position in a d-dimensional space.

- Consider two-dimensional case:

Computing a node's position in a two-dimensional space.



Just as in GPS, node P can compute is own coordinates (x_P,y_P) by solving the three equations with the two unknowns x_P and y_P:

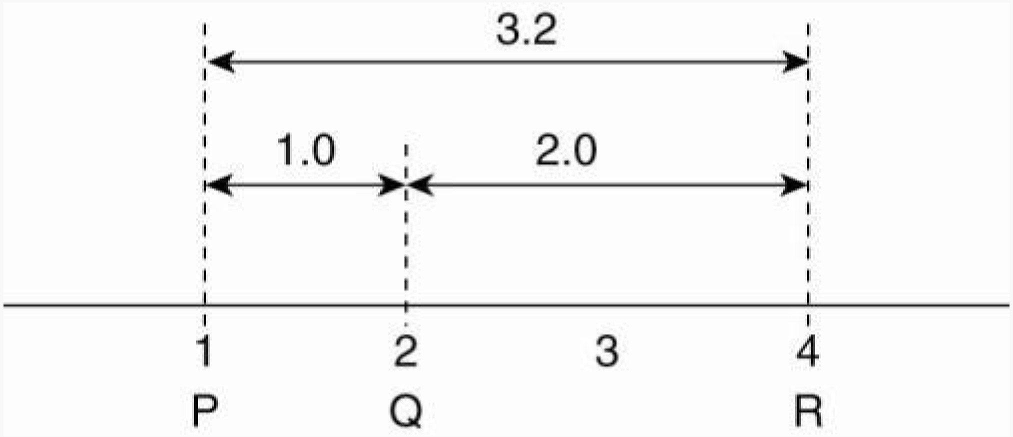
$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2} \qquad (i=1,2,3)$$

- d_i corresponds to measuring the latency between P and the node at (x_i,y_i).
- Latency estimated as half the round-trip dela.

Problems:

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

Inconsistent distance measurements in a one-dimensional space.



Solution (Ng and Zhang 2002)**:** 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 $\hat{d}(b_i,b_j)$ denotes the distance to landmark b_i given a computed coordinate for P.

Observation:

With well-chosen landmarks, m can be as small as 6 or 7, with $\hat{d}(P,Q)$ being no more than a factor 2 different from the actual latency $d(P,Q)$ for arbitrary nodes P and Q (Szyamniak et al., 2004).

Another Approach:

- Vivaldi: optimizes the network as a collection of nodes connected via springs.
- It can be shown that the system will eventually converge to an optimal organization in which the aggregated error is minimal.
- This approach is followed in, of which the details can be found in Dabek et al. (2004a).

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 => single point of failure.

Election by Bullying (Garcia-Molina 1982):

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 $P\text{light}$, 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.

The bully election algorithm:

- The group consists of eight processes, numbered from 0 to 7.
- Previously process 7 was the coordinator, but it has just crashed.

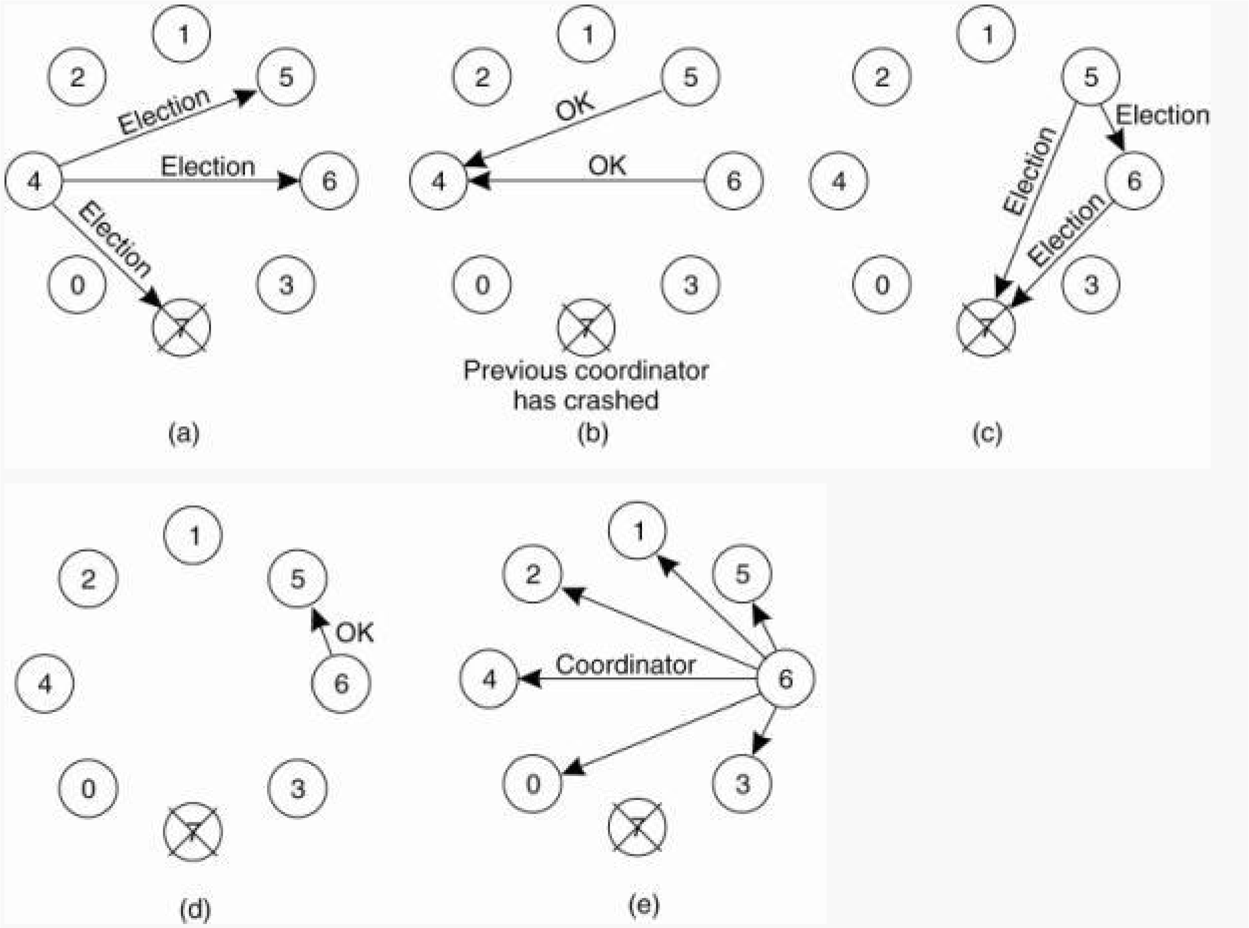
(a) Process 4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely 5, 6, and 7,

(b) Processes 5 and 6 respond, telling 4 to stop.

(c) Now 5 and 6 each hold an election.

(d) Process 6 tells 5 to stop.

(e) Process 6 wins and tells everyone.



(a) (b) (c) (d) (e)

NOTE:

- If a process that was previously down comes back up, it holds an election.
- If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job.
- Thus the biggest guy in town always wins, hence the name "bully algorithm."

Election in a Ring

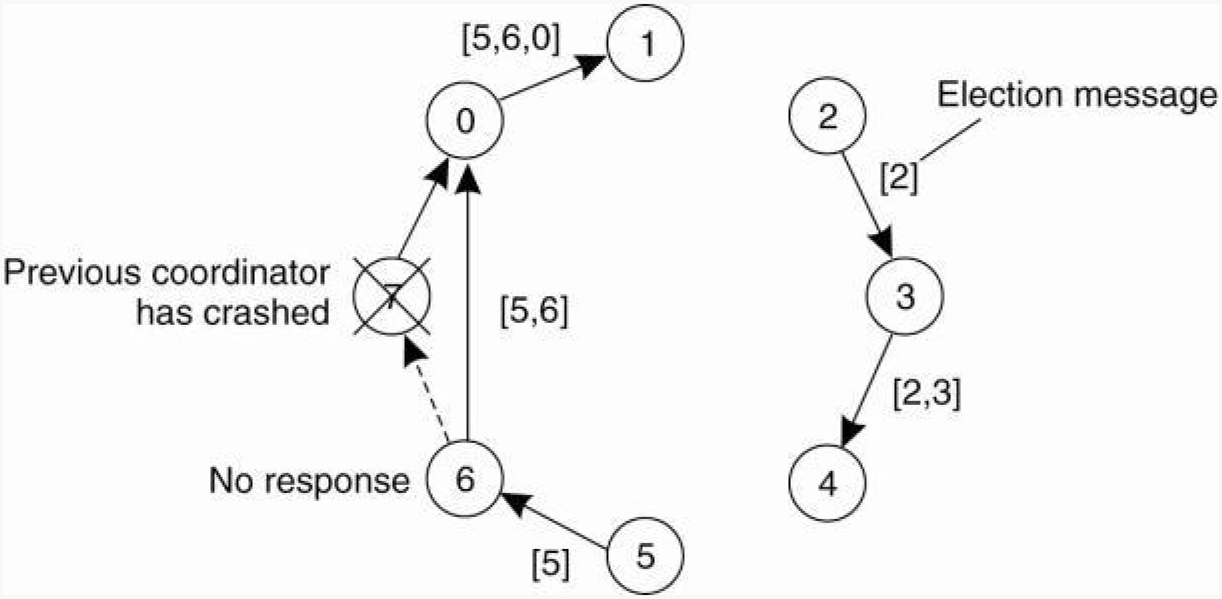
Principle: Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- 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.

Example:

- What happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed.
- Each builds an ELECTION message and each starts circulating its message, independent of the other.
- Both messages will complete a circuit, and both 2 and 5 will convert them into COORDINATOR messages, with exactly the same members and in the same order.
- When both have completed a second circuit, both will be removed.



Elections in Large-Scale Systems

Superpeer Election (Lo et al. 2005)

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 a predefined fraction of superpeers
- Each superpeer should does not need to serve more than a fixed number of normal nodes

DHT: Reserve a fixed part of the ID space for superpeers.

- In DHT-based systems, the basic idea is to reserve a fraction of the identifier space for superpeers.
- In DHT-based systems each node receives a random and uniformly assigned *m*-bit identifier.
- Now reserve the first (i.e., leftmost) k bits to identify superpeers.

Example: if S superpeers are needed for a system that uses m-bit identifiers, simply reserve the $k = [\log_2 S]$ 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 $11\cdots1100\cdots00$

Example:

- A (small) Chord system with m = 8 and k = 3.
- When looking up the node responsible for a specific key p, we can first decide to route the lookup request to the node responsible for the pattern

p AND 11100000

which is then treated as the superpeer.

- Note that each node id can check whether it is a superpeer by looking up

id AND 11100000

to see if this request is routed to itself.