Synchronization

Today

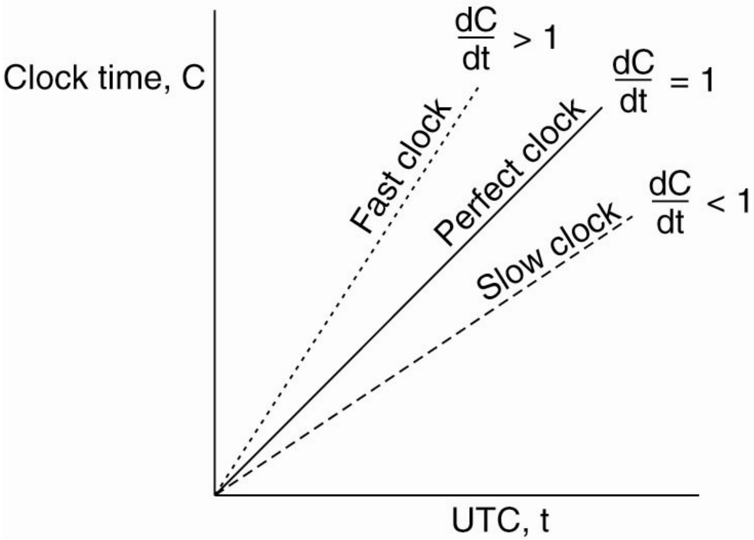
- Physical and Logical clocks
- Mutual exclusion
- Election algorithms

Physical clocks

- Sometimes we need the exact time
- 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 & satellite. Satellites can give an accuracy of about ±0.5 ms.
- We want to distribute this to a bunch of machines
  - Each runs its own timer, keeping a clock $C_p(t)$ (t being UTC)
  - Ideally we want $C_p(t) = t$ for all processes, i.e. $dC/dt = 1$

Physical clocks

- However, $1 - r \leq dC/dt \leq 1 + r$



- Goal: Never let two clocks in any system differ by more than *d time units* ⇒ *synchronize at least every d/(2r) seconds.*

Clock synchronization

- Model 1 – Every machine asks a time server for the accurate time at least once every *d/(2r) seconds* (Network Time Protocol)
  - You need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.
- Model 2 – 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 you don't even need to propagate UTC time.
- You'll have to take into account that setting the time back is never allowed ⇒ smooth adjustments

Happened-before relationship

- 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* $a \rightarrow b$.
  - *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.

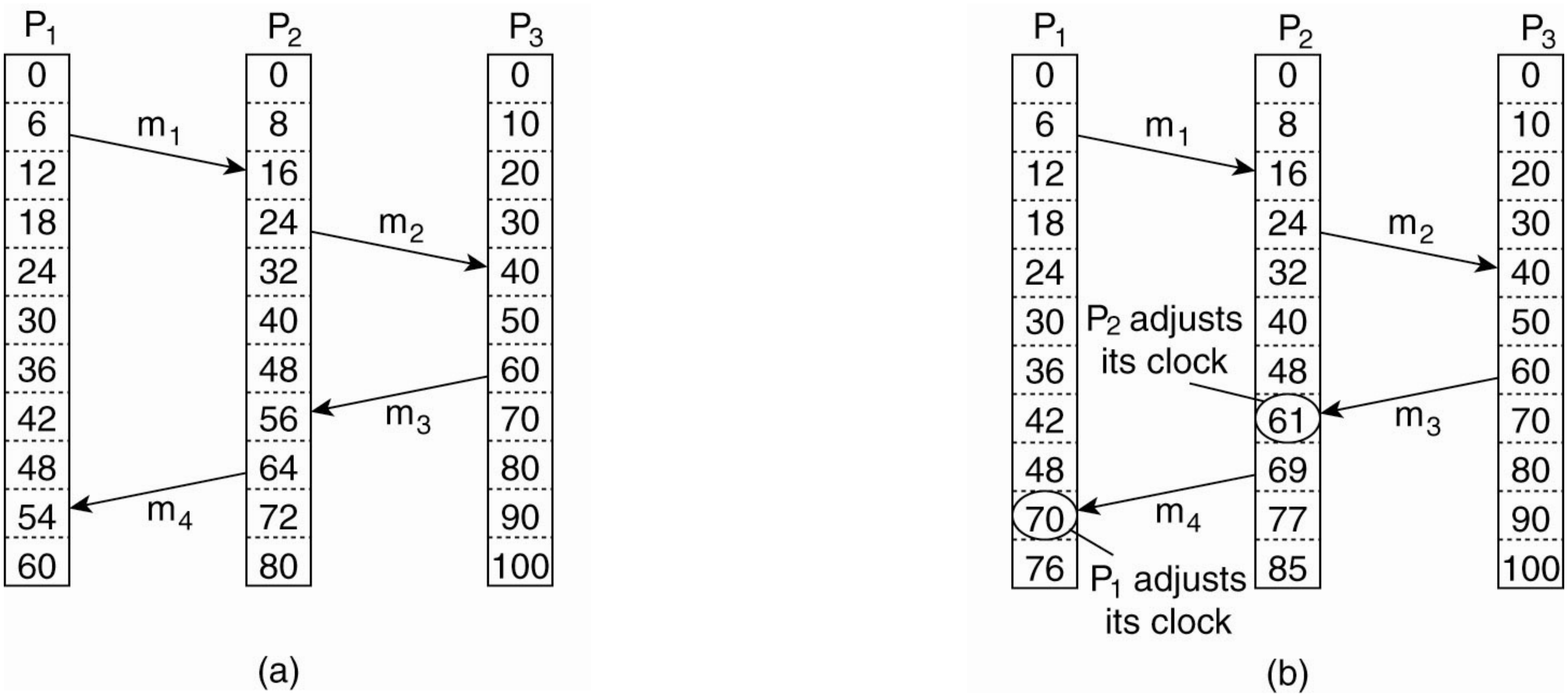
Lamport clock

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

Lamport clock

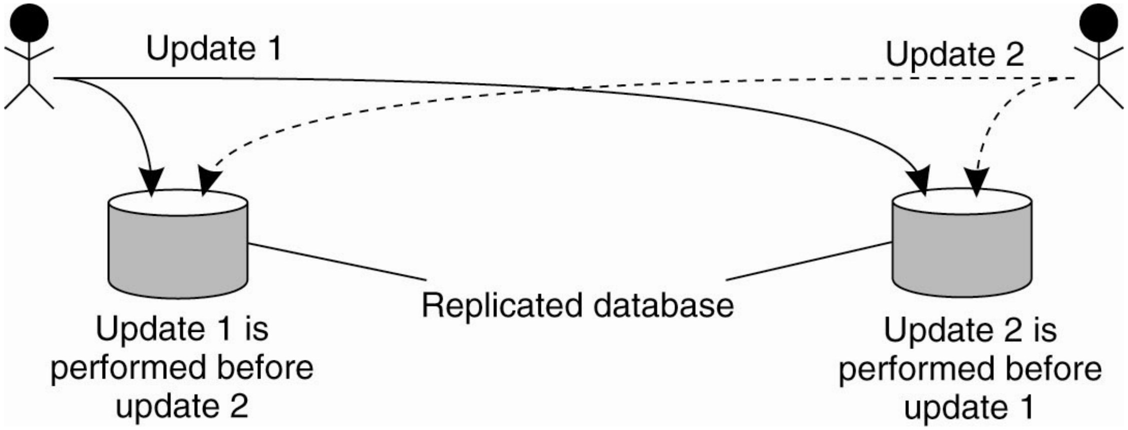
- Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:
  - 1: For any two successive events that take place within P_i, C_i is incremented by 1.
  - 2: Each time a message m is sent by process P_i, the message receives a timestamp ts(m) = C_i.
  - 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.
- Property 1 is satisfied by (1);
- Property 2 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.

Lamport clock - an example



Example use – totally ordered multicast

- We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:
  - *P1* adds $100 to an account (initial value: $1000)
  - *P2* increments account by 1%
  - There are two replicas



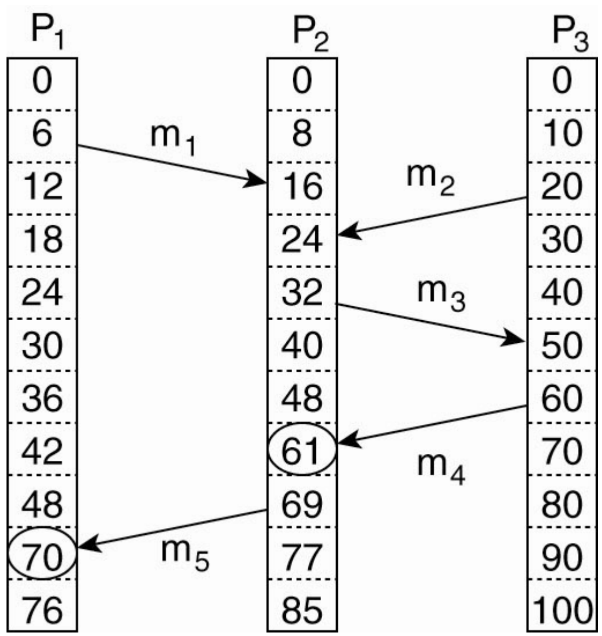
Result: in absence of proper synchronization: replica #1 ← $1111, while replica #2 ← $1110.

Totally ordered multicast

- Solution:
  - Process *Pi 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_3 is sent at $T = 32$
  - The sending of m_3 may have been affected by m_1



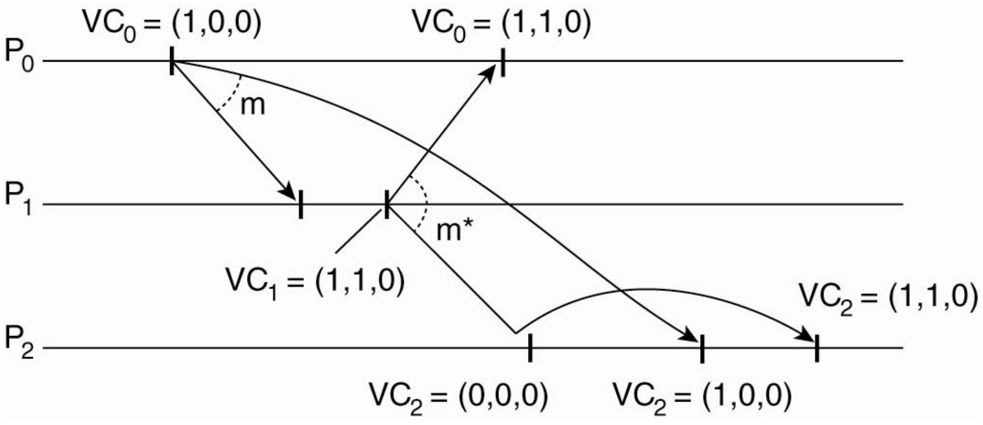
- But,
  - Event *a:* m_1 is received at $T = 16$
  - Event *b:* m_2 is sent at $T = 20$
  - **We cannot conclude that a causally precedes *b***

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?

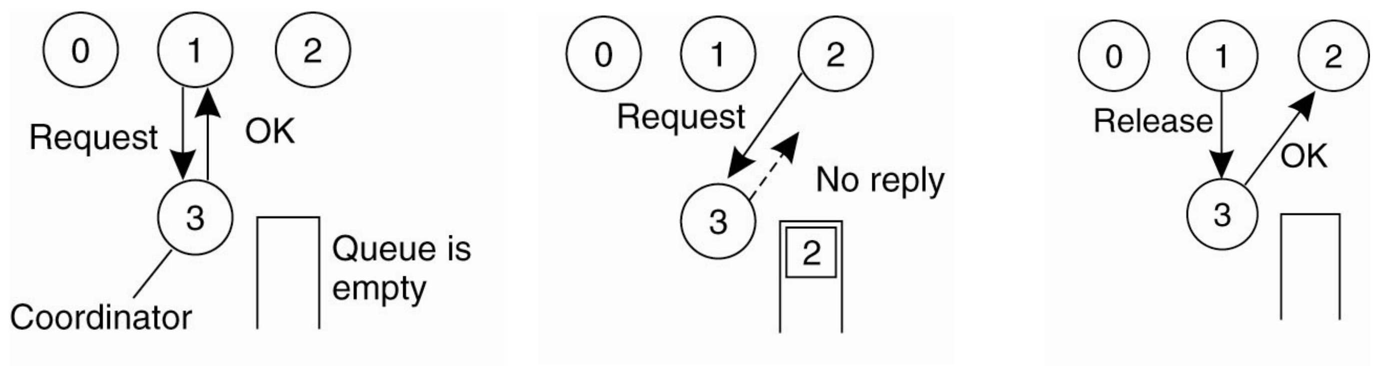
Causally ordered multicasting

- We can now ensure that a msg is delivered only if all causally preceding msgs have already been delivered
- Adjustment: *Pi* 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 $k \mathrel{!}= j$



Mutual exclusion

- Processes 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:
  - Good – It works, is easy to implement; takes few messages
  - Bad – Central point of failure & potential bottleneck

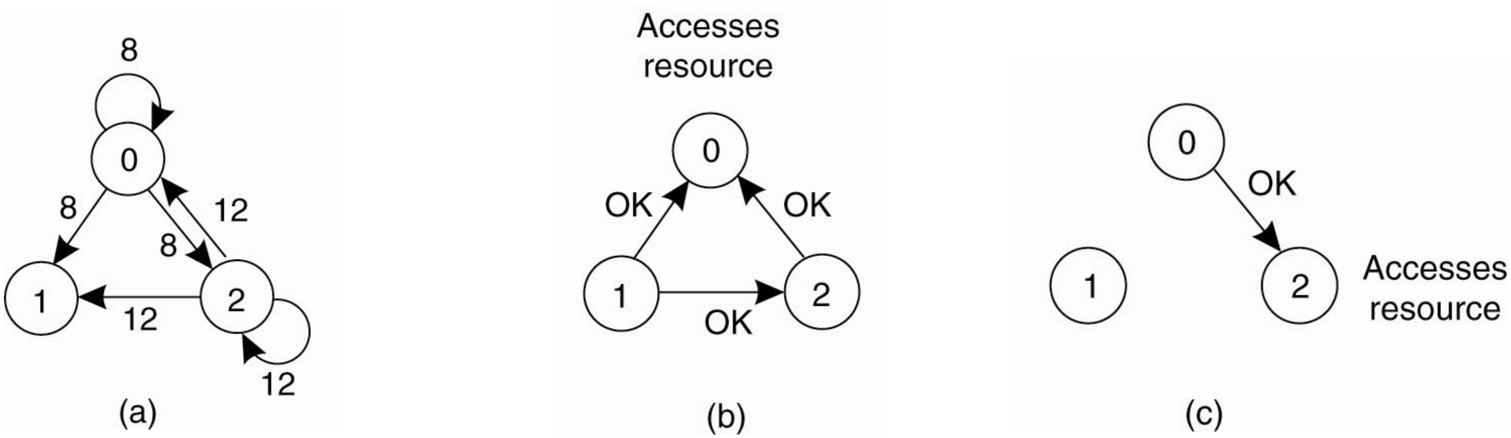


Decentralized algorithm

- 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 (either way)
- Assumption – When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted
- Good – Very low probability of violating correctness
- Bad – With high contention may come low utilization

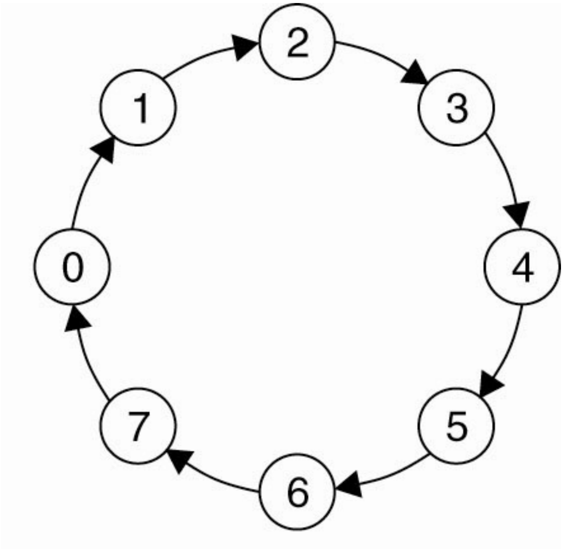
Distributed algorithm

- 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 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.



Token-based

- 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)



Comparing the different 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

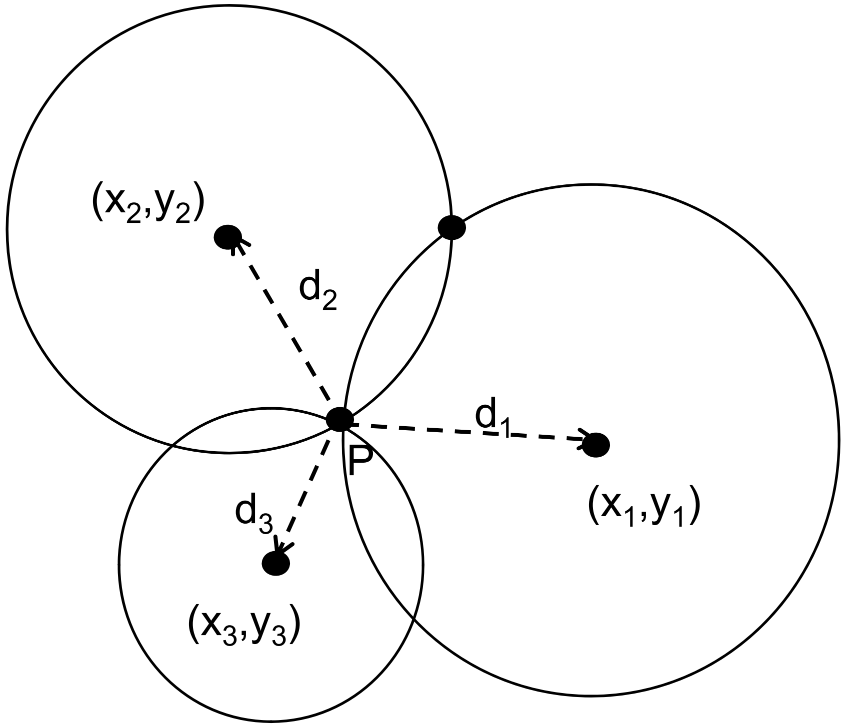
Global positioning of nodes

- How can a single node efficiently estimate the latency between any two other nodes in a distributed system?
- Construct a geometric overlay network, in which the distance *d(P,Q)* reflects the actual latency between *P* and *Q*.

A node P needs k + 1 landmarks to compute its own position in a d-dimensional space

In 2d, P needs to solve three equations in two unknowns (x_P, y_P):

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



Global positioning of nodes

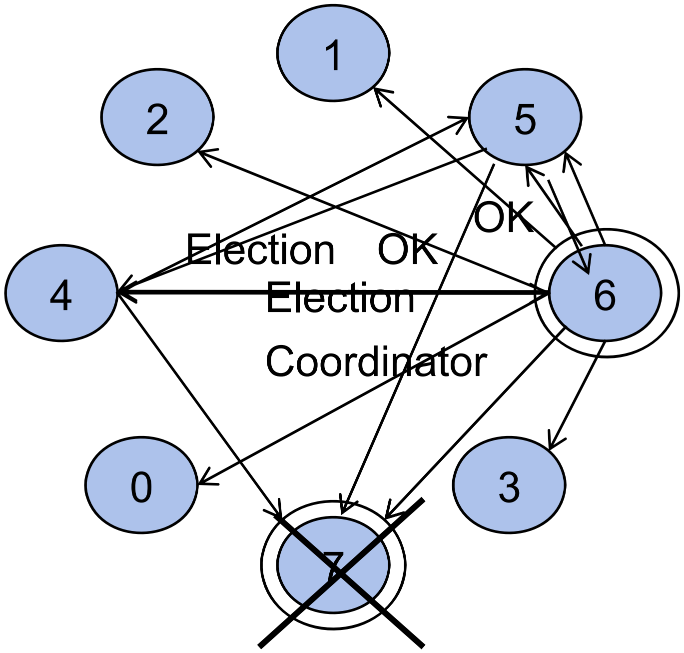
- d_i generally corresponds to latency, estimated as half the round-trip delay
- But latency changes over time, and “error” propagates
- Considering that Internet latency generally violates the triangle inequality ($d(P,R) \leq d(P,Q) + d(Q,R)$) it’s generally impossible to fix all inconsistencies
- A few ways to address this
  – Use special nodes, landmarks, and compute coordinates to minimize aggregated errors (GNP)
  – See networks as nodes connected by springs, the error being their relative displacement from rest (Vivaldi)
  – Avoid embedding errors with direct measurement (Meridian)
  – Reuse the network view of others, such as CDNs (CRP)

Election algorithms

- Many distributed algorithms require one process to act as coordinator
- In general, it doesn't matter which one – so pick the one with the largest ID/weight
- We assume every process knows the identity of all other processes, just not who is up/down
- Elections conclude when all agree on new coordinator

The Bully algorithm

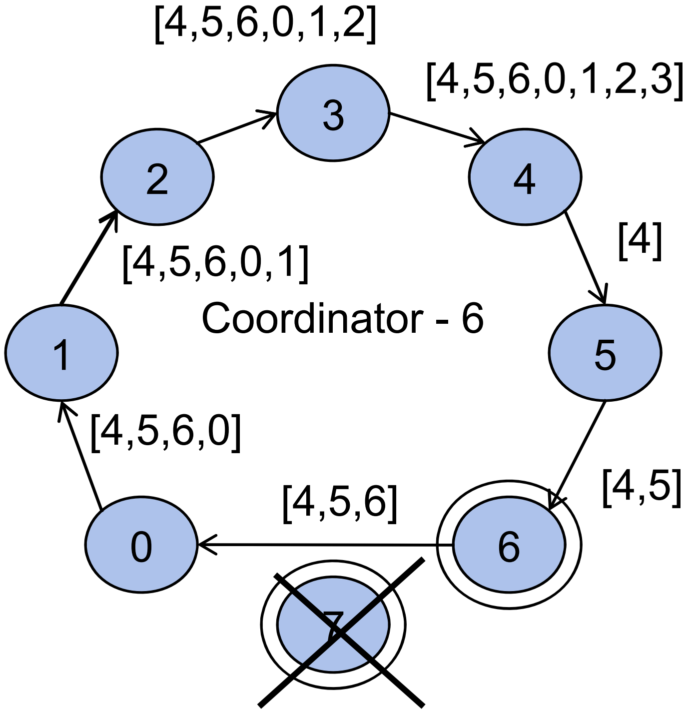
- Somebody, P, notice coordinator is down and calls an election
- P sends ELECTION message to all processes with higher numbers
- If no-one responds, P is the winner
- If a process with a higher number receives the ELECTION message, reply with OK and calls an election
- When done, winner let everybody know with a COORDINATOR message
- If 7 ever wakes up, it will call for elections



Garcia-Molina, '82

A ring algorithm

- Somebody, P, notice coordinator is down and calls an election
- P sends ELECTION message with its number in to first successor up
- Recipient forward messages adding itself as candidate
- Who started it all, will eventually receive a message with itself in the list; elect coordinator and inform all
- COORDINATOR messages goes around the ring once

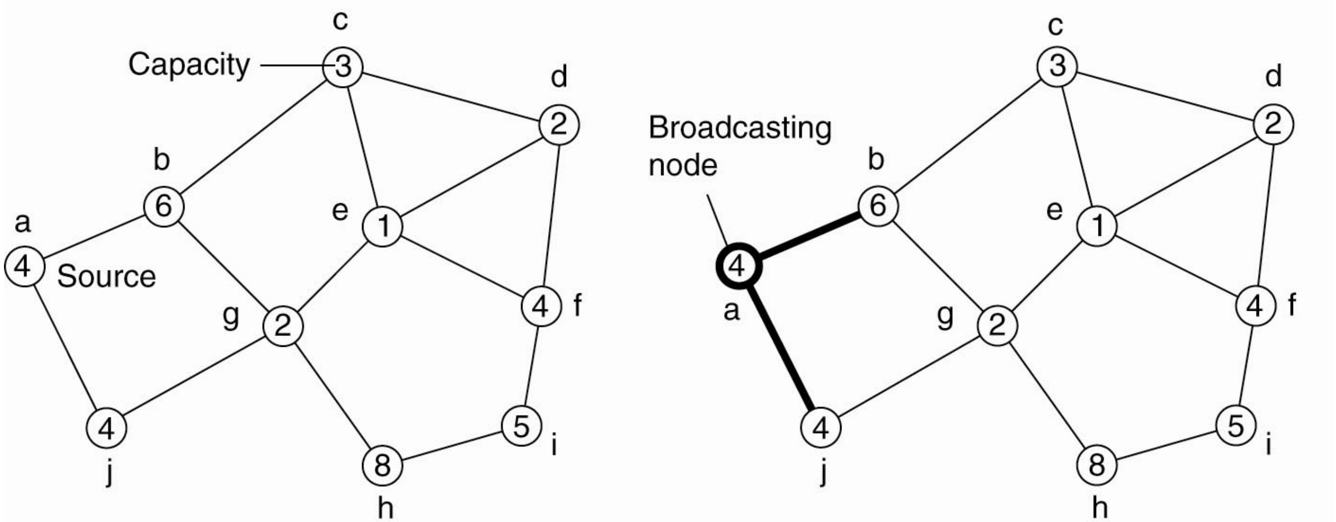


Election in large-scale systems

- Electing superpeers in a P2P system; requirements
  - Normal nodes should have low latency access to superpeers
  - Superpeers should be evenly distributed through the overlay
  - There should be a predefined % of superpeers
  - Each superpeer should serve no more than a fix # of normal peers
- In a DHT-based system, pick the first k bits to identify a superpeer; if S superpeers, $k = \lceil \log_2 S \rceil$
  - Need to route to node responsible for key p? (With k = 3) Go to p AND 111000…
- To position N nodes evenly in a m-dim space
  - Distribute N tokens to randomly nodes; tokens repel each other; use gossiping to disseminate tokens' forces; holder is superpeer

Election in wireless environments

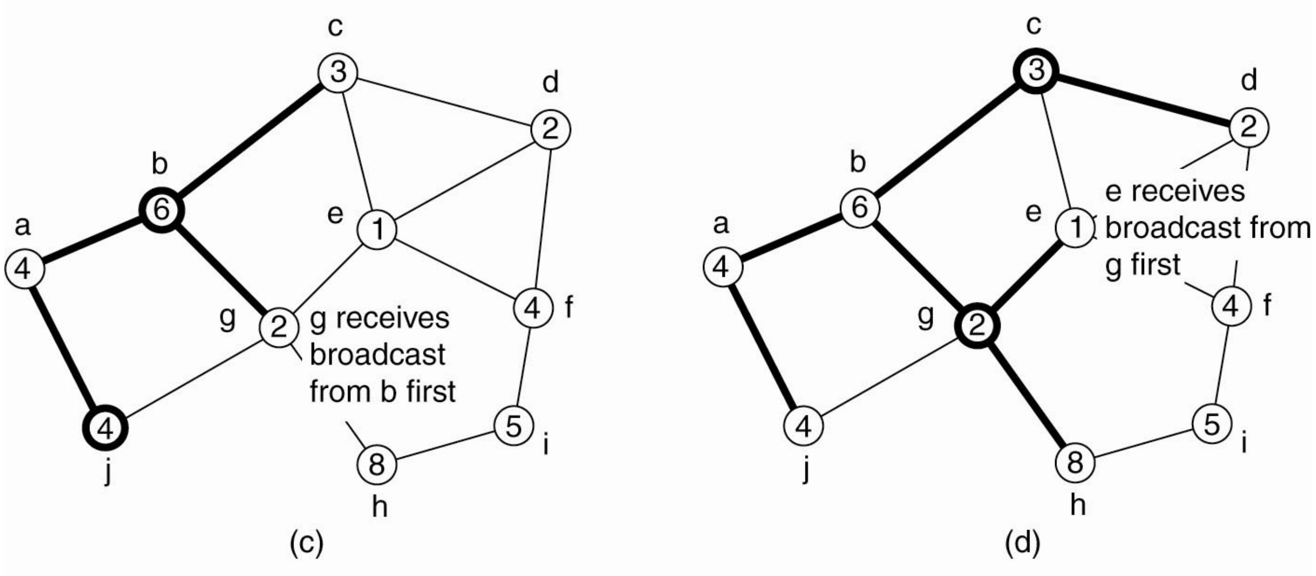
- Traditional algorithms make assumptions not realistic in wireless settings (e.g. message passing is reliable)
- Elect the “best” leader based on dynamic tree construction
- Election messages are tagged with unique ID to deal with concurrent elections



(a) (b)

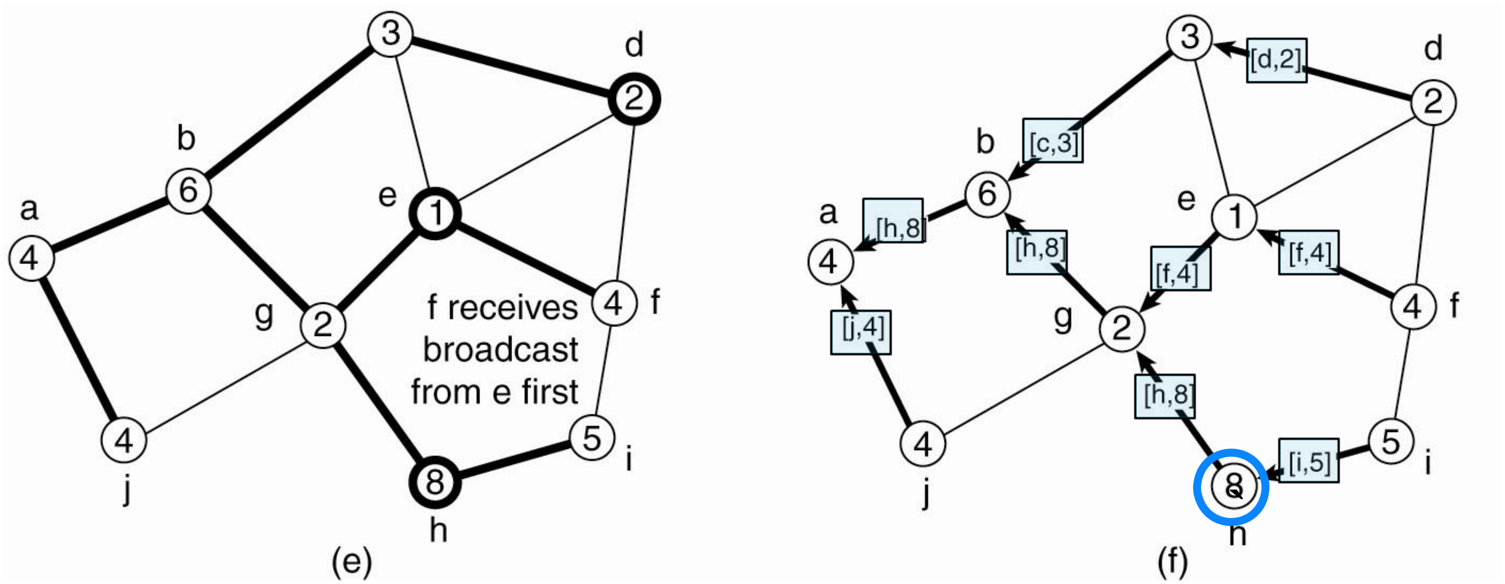
Election in wireless environments

- When a node receives an election message for the first time, it select source as parent and forwards the message



Election in wireless environments

- Leaf nodes report to parent with their capacity
- Children pass the most eligible node up the tree



Summary

- Synchronization is about doing the right thing at the right time …
- What’s the right time?
  - An issue when you don’t share clocks
- What’s the right thing to do?
  - Who can access what when?
  - Who is in charge?