Today: More Classical Problems

Leader election

Mutual exclusion

Election Algorithms

- Many distributed algorithms need one process to act as coordinator
  - Doesn't matter which process does the job, just need to pick one
- Election algorithms: technique to pick a unique coordinator (aka *leader election*)
- Examples: take over the role of a failed process, pick a master in Berkeley clock synchronization algorithm
- Types of election algorithms: Bully and Ring algorithms

Bully Algorithm

- Each process has a unique numerical ID
- Processes know the IDs and address of every process
- Communication is assumed to be reliable
- *Key Idea*: select process with highest ID
- *Challenges:*
  - Fault tolerance. *Why?*
  - Several processes can initiate an election simultaneously. *Why?*
- *Simplest solution?*

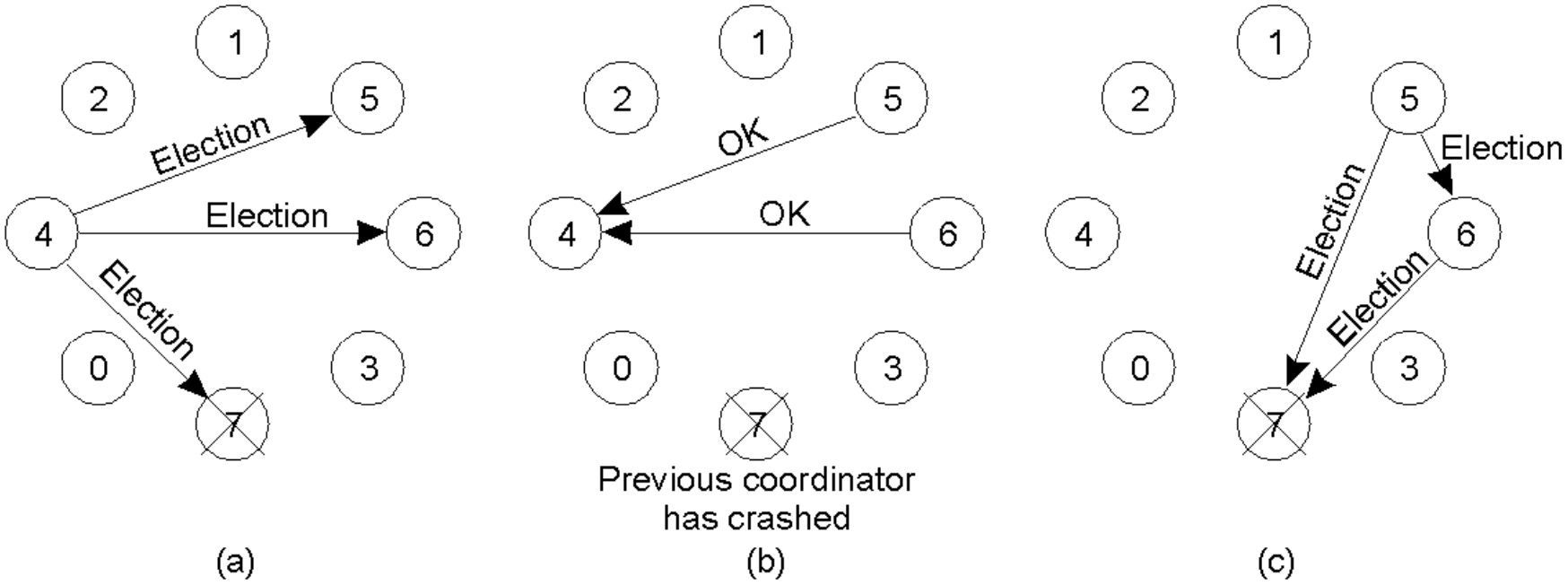
Bully Algorithm Details I

- Any process *P* can initiate an election
  - Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *Election, OK, Coordinator*
- $O(n^2)$ messages required with n processes

Bully Algorithm Details II

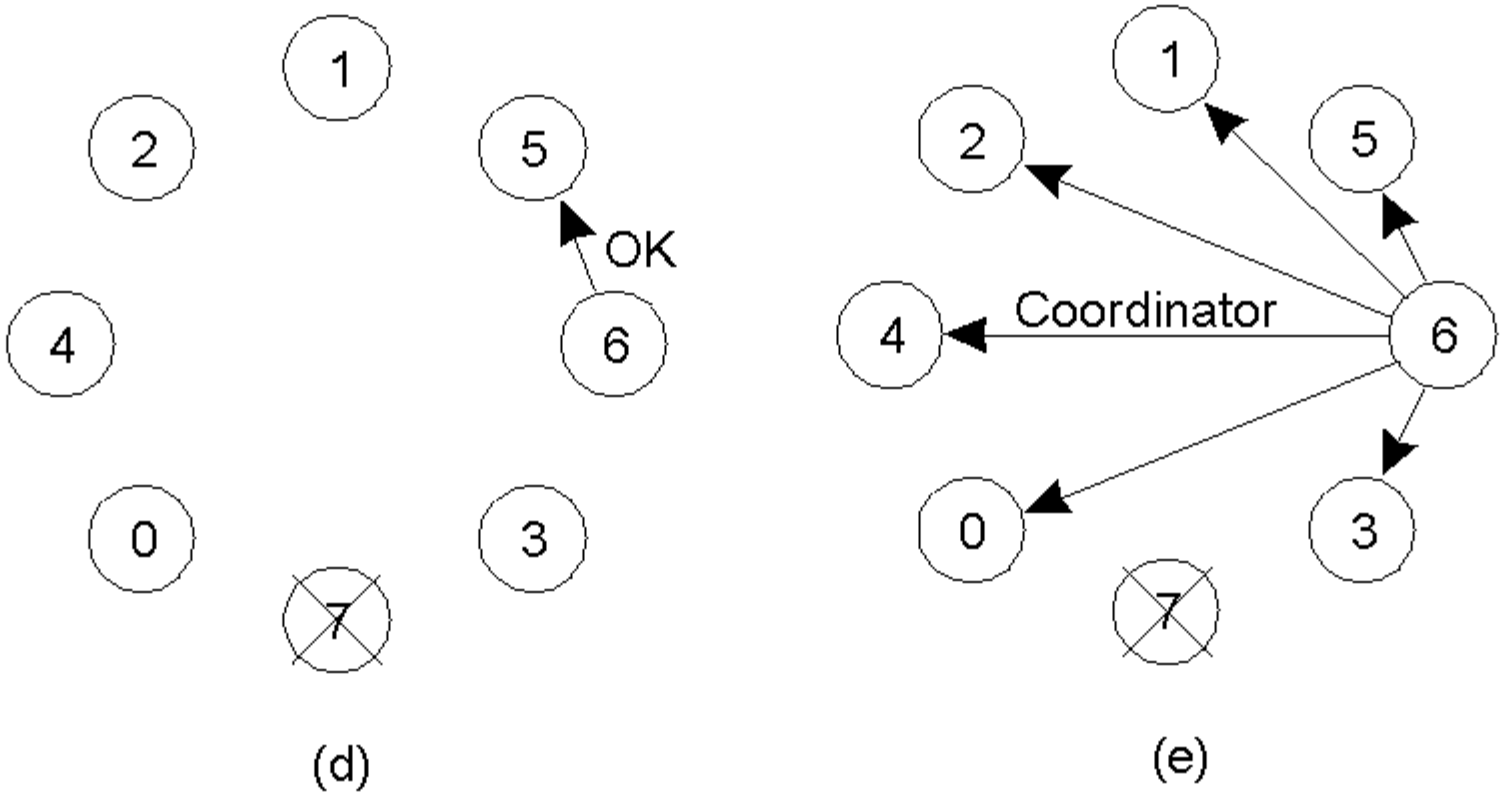
- *P* sends *Election* messages to all process with higher IDs and awaits *OK* messages
  - If no *OK* messages, *P* becomes coordinator and sends *Coordinator* messages to all process with lower Ids
  - If it receives an *OK*, it drops out and waits for a *Coordinator*
- If a process receives an *Election* msg, it returns an *OK* and starts an election
- If a process receives a *Coordinator*, it treats sender an coordinator

Bully Algorithm Example



- The bully election algorithm
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

Bully Algorithm Example



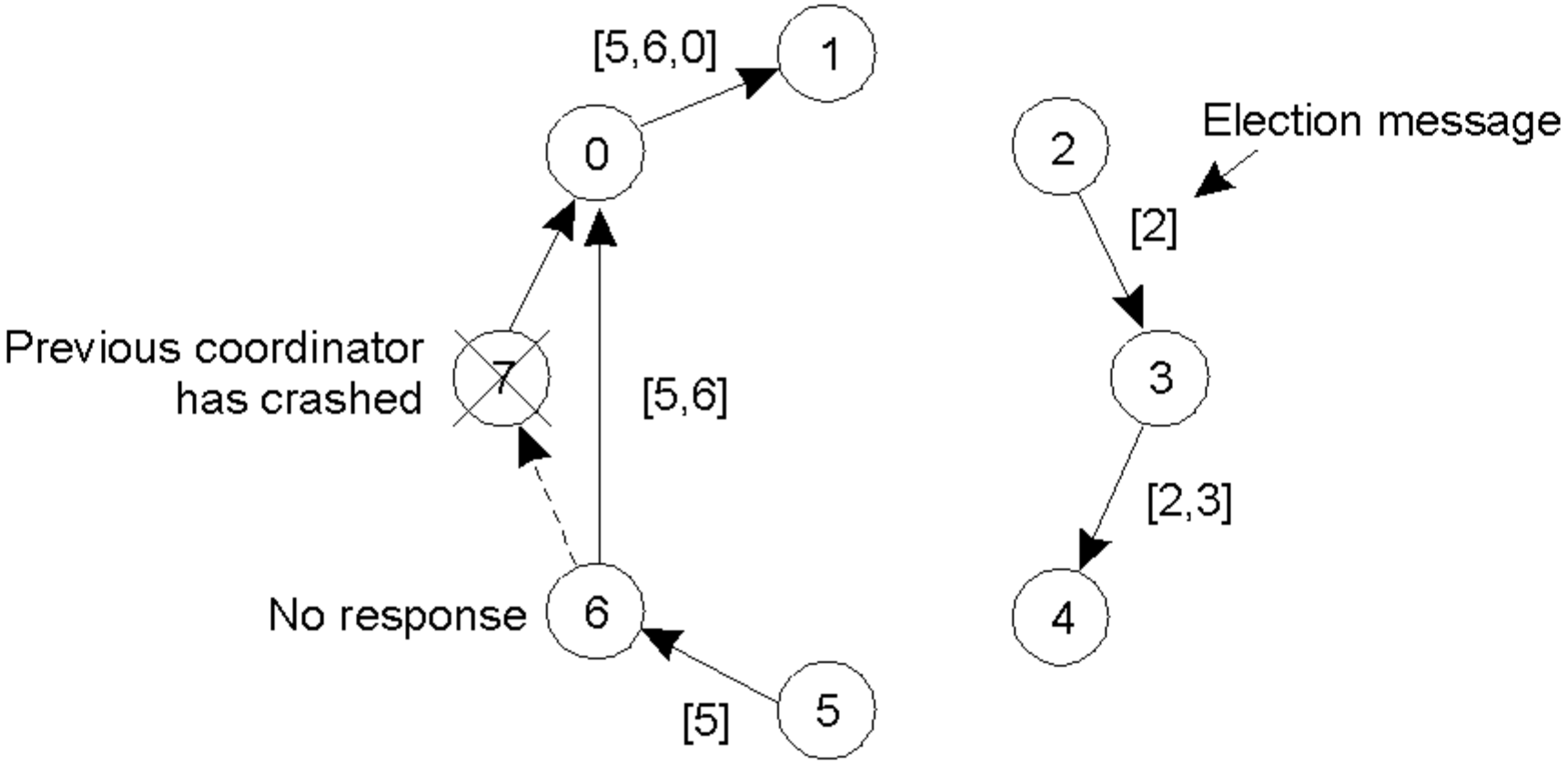
d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone

Ring-based Election

- Processes have unique IDs and are arranged in a logical ring
  - Each process knows its neighbors
- Like in the previous algorithm
  - Select process with highest ID
  - Begin election if just recovered or coordinator has failed
- Send *Election* to closest downstream node that is alive
  - Sequentially poll each successor until a live node is found
  - Each process tags its ID on the message
- *Initiator* picks node with highest ID and sends a *Coordinator* message
- Multiple elections can be in progress
  - Wastes network bandwidth but does no harm

A Ring Algorithm



Example with two initiators

Eventually both initiators agree on leader

Synchrony Assumptions

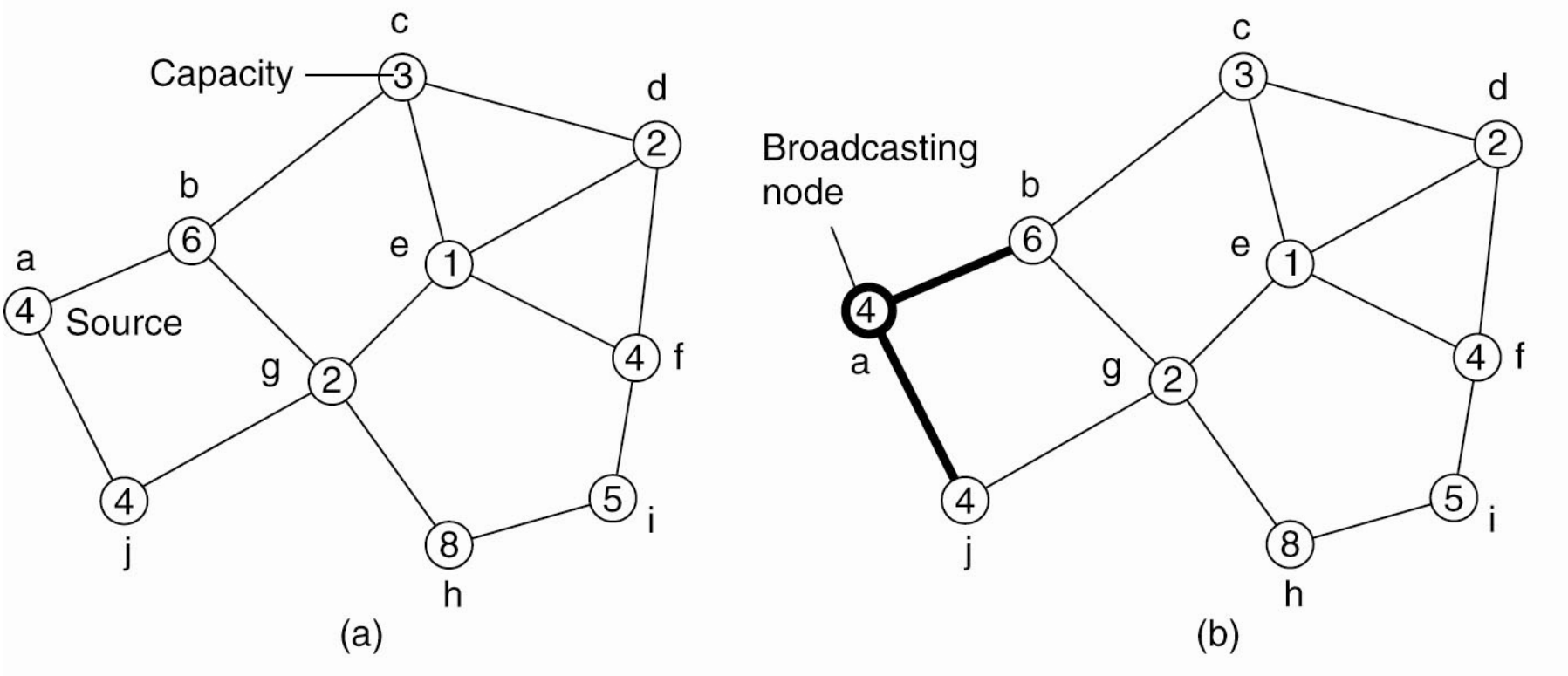
- These protocols make *synchrony assumptions*
  - E.g. the timers used to detect faults are accurate
  - These don't always hold
  - If synchrony does not hold => there may not be an agreement on a leader
- In general, agreement in an *asynchronous* systems is *not* possible (more on this later in the course)

Comparison

- Assume *n* processes and one election in progress

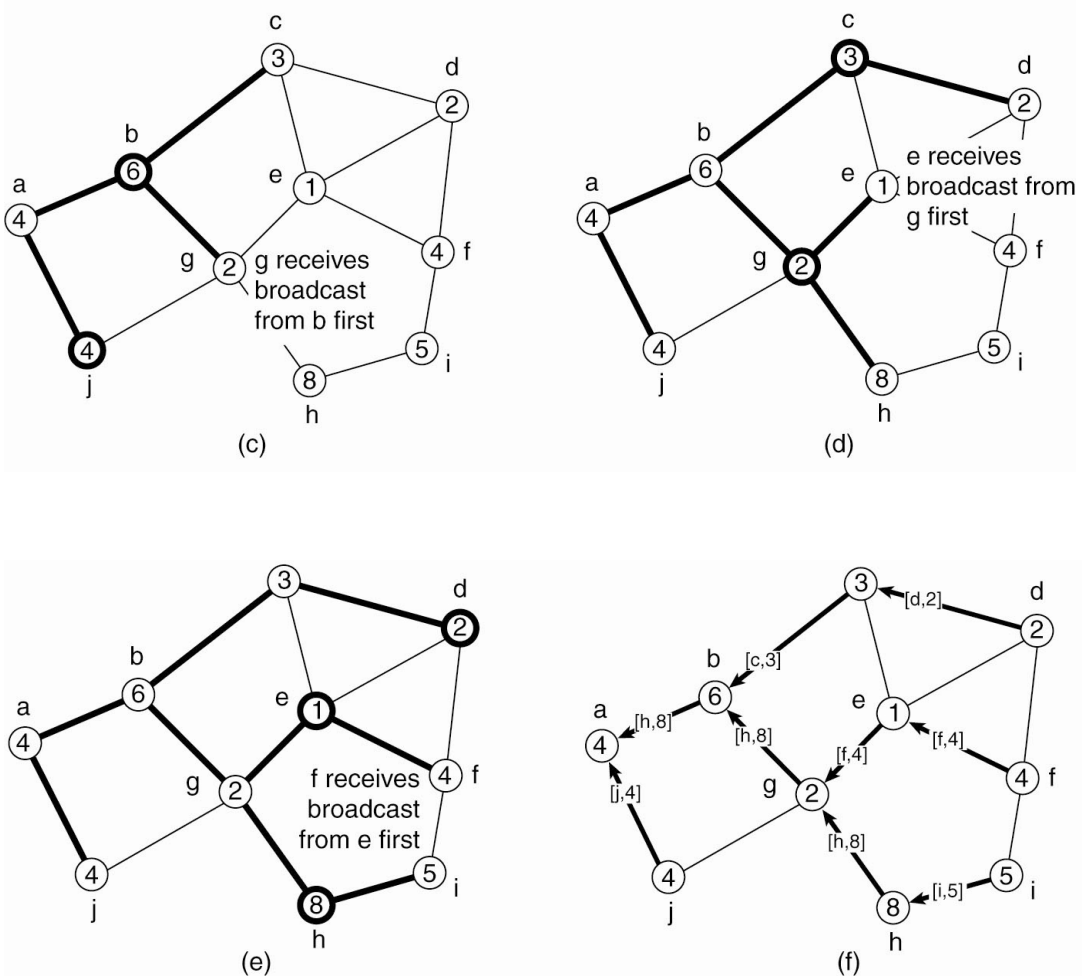
- Bully algorithm
  - Worst case: initiator is node with lowest ID
    - Triggers *n-2* elections at higher ranked nodes: $O(n^2)$ msgs
  - Best case: immediate election: *n-2* messages
- Ring
  - *2 (n-1)* messages always

Elections in Wireless Environments I



- Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase

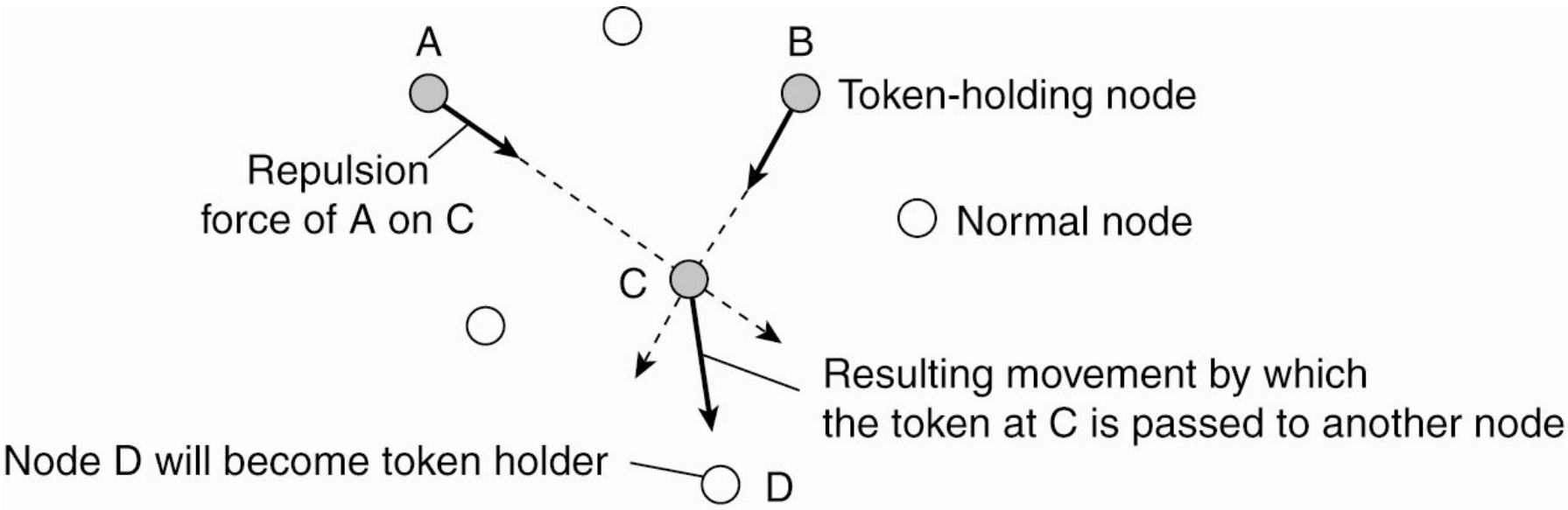
Elections in Wireless Environments II



Elections in Large-Scale Systems

- Variant: superpeer election in overlay networks
  1. Normal nodes should have low-latency access to superpeers.
  2. Superpeers should be evenly distributed across the overlay network.
  3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
  4. Each superpeer should not need to serve more than a fixed number of normal nodes.

Elections in Large-Scale Systems (2)



- Moving tokens in a two-dimensional space using repulsion forces.

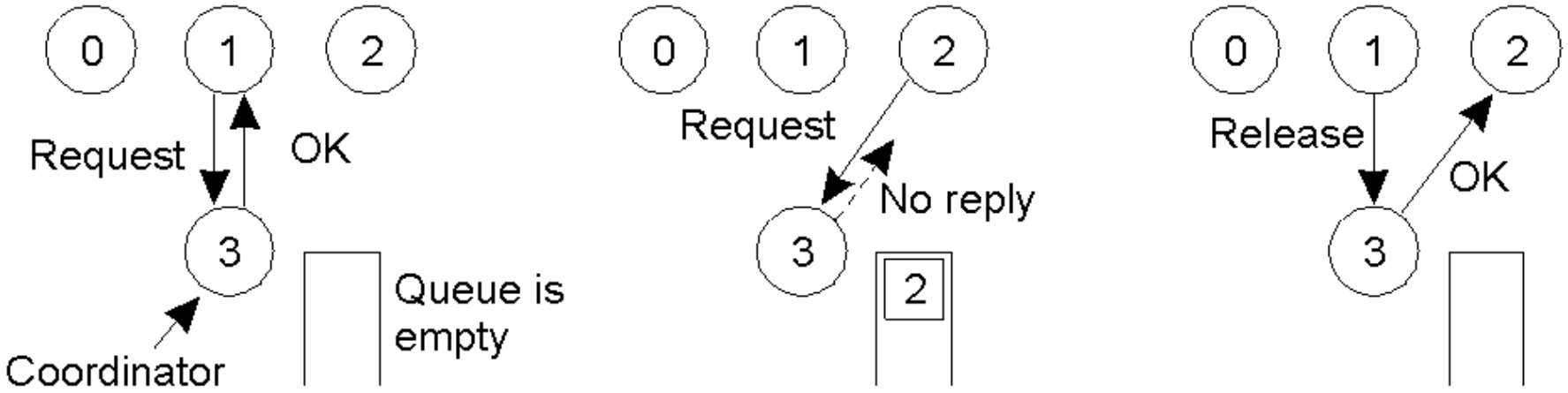
Distributed Synchronization

- Distributed system with multiple processes may need to share data or access shared data structures
  - Use critical sections with mutual exclusion
- Single process with multiple threads
  - Semaphores, locks, monitors
- How do you do this for multiple processes in a distributed system?
  - Processes may be running on different machines
- Solution: lock mechanism for a distributed environment
  - Can be centralized or distributed

Centralized Mutual Exclusion

- Assume processes are numbered
- One process is elected coordinator (highest ID process)
- Every process needs to check with coordinator before entering the critical section
- To obtain exclusive access: send request, await reply
- To release: send release message
- Coordinator:
  - Receive *request*: if available and queue empty, send *OK*; if not, queue request
  - Receive *release*: remove next request from queue and send *OK*

Mutual Exclusion: A Centralized Algorithm



(a) (b) (c)

a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

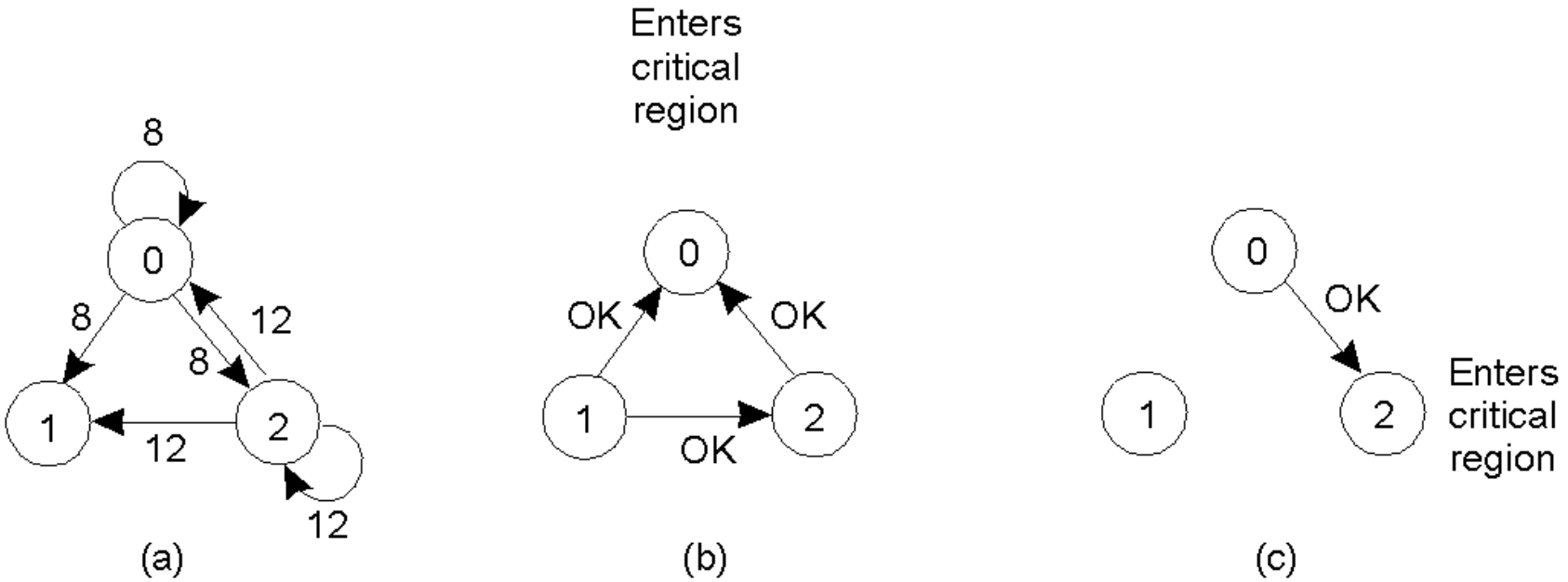
Properties

- Simulates centralized lock using blocking calls
- Fair: requests are granted the lock in the order they were received
- Simple: three messages per use of a critical section (request, grant, release)
- Shortcomings:
  - Single point of failure
  - How do you detect a dead coordinator?
    - A process can not distinguish between "lock in use" from a dead coordinator
      - No response from coordinator in either case
  - Performance bottleneck in large distributed systems

Distributed Algorithm

- Based on event ordering and time stamps
  - Assumes total ordering of events in the system (Lamport's clock)
- Process *k* enters critical section as follows
  - Generate new time stamp $TS_k = TS_k + 1$
  - Send *request*(k, TS_k) all other *n-1* processes
  - Wait until *reply(j)* received from all other processes
  - Enter critical section
- Upon receiving a *request* message from process *k*, process *j*
  - Sends *reply* if no contention
  - If *j* already in critical section, does not reply, queue request
  - If *j* wants to enter, compare TS_j with TS_k and send reply if $TS_k < TS_j$, else queue (recall: total ordering based on multicast)
- *Question: starvation?*

A Distributed Algorithm



a) Two processes want to enter the same critical region 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 enter the critical region.

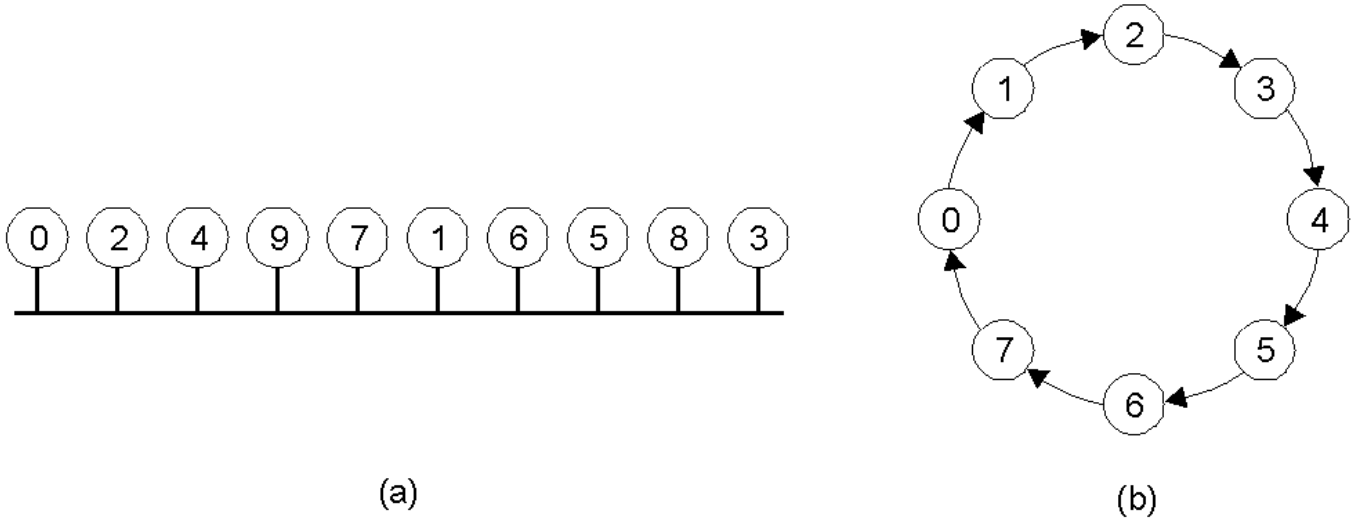
Properties

- Fully distributed
- *N* points of failure!
- All processes are involved in all decisions
  - Any overloaded process can become a bottleneck

Decentralized Algorithm

- Use voting
- Assume *n* replicas and a coordinator per replica
- To acquire lock, need majority vote $m > N/2$ coordinators
  - Non-blocking: coordinators returns *OK* or *NO*
- If a coordination crashes => it forgets previous votes
  - Assume one crash during an interval has probability *p*
  - Probability that exactly *k* coordinators crash during interval
    - $P(k) = C^m_k \, p^k \, (1\text{-}p)^{m-k}$
  - At least *N/2* must crash to violate correctness
    - $\sum^m_{k=m-N/2} P(k)$

A Token Ring Algorithm



a) An unordered group of processes on a network.

b) A logical ring constructed in software.

- Use a token to arbitrate access to critical section
- Must wait for token before entering CS
- Pass the token to neighbor once done or if not interested
- Detecting token loss in non-trivial

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk	2m	starvation
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

- A comparison of four mutual exclusion algorithms.