Last Class

- Vector timestamps
- Global state
  - Distributed Snapshot
- Election algorithms

Today: Still More Canonical Problems

- Election algorithms
  - Bully algorithm
  - Ring algorithm
- Distributed synchronization and mutual exclusion
- Distributed transactions

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

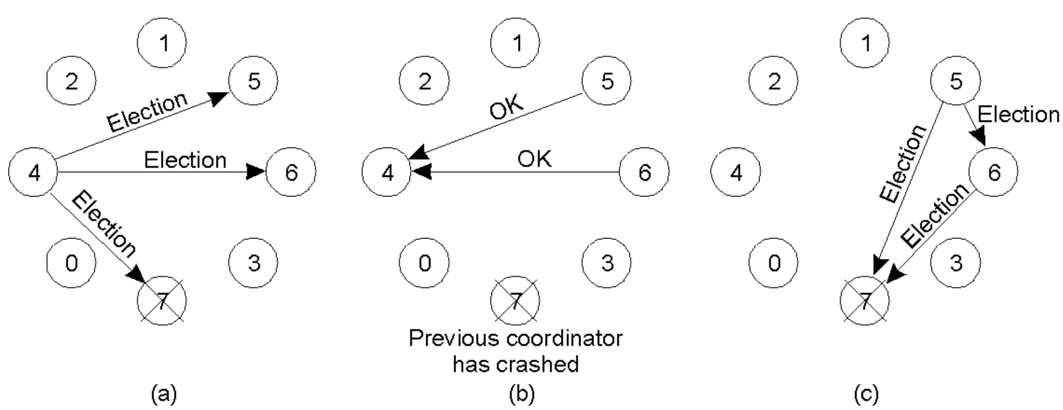
Bully Algorithm

- Each process has a unique numerical ID
- Processes know the Ids and address of every other process
- Communication is assumed reliable
- *Key Idea*: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *election, OK, I won*
- Several processes can initiate an election simultaneously
  - Need consistent result
- $O(n^2)$ messages required with n processes

Bully Algorithm Details

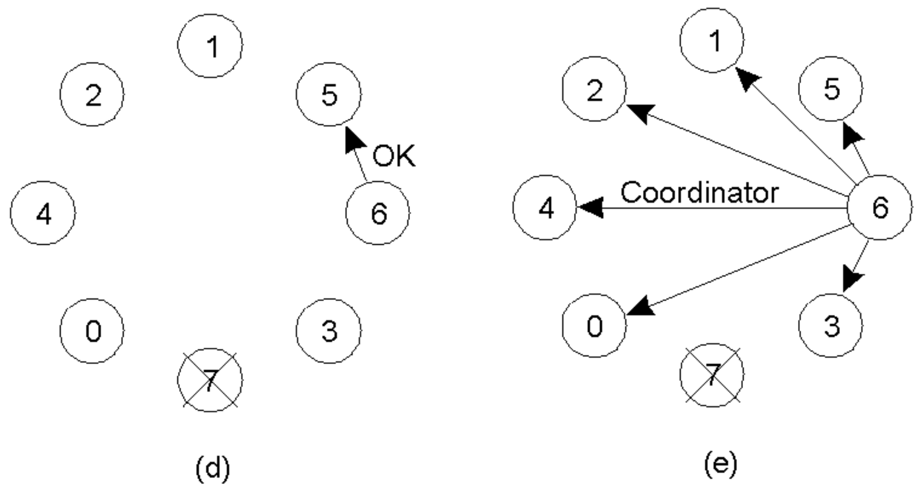
- Any process *P* can initiate an election
- *P* sends *Election* messages to all process with higher Ids and awaits *OK* messages
- If no *OK* messages, *P* becomes coordinator and sends *I won* messages to all process with lower Ids
- If it receives an *OK*, it drops out and waits for an *I won*
- If a process receives an *Election* msg, it returns an *OK* and starts an election
- If a process receives a *I won*, 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

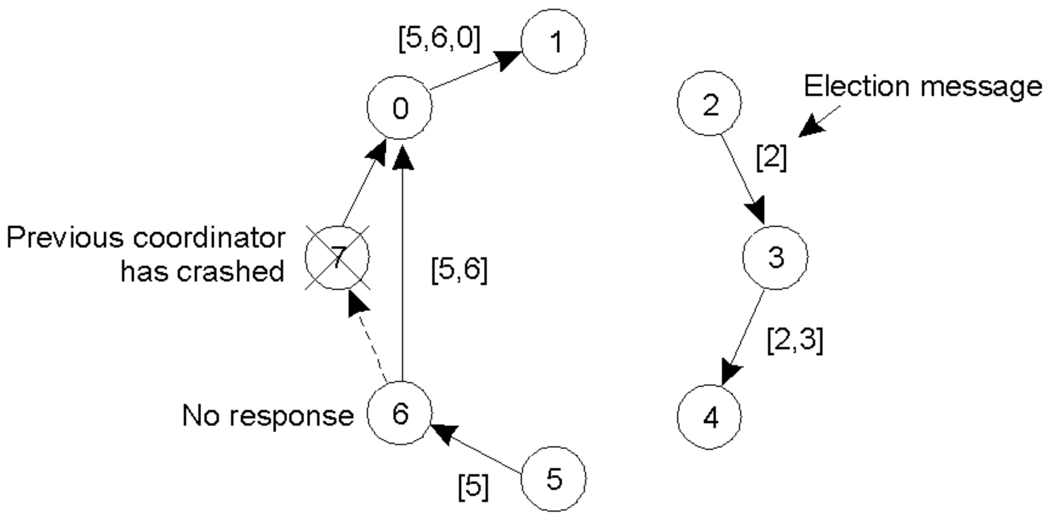


d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone

Ring-based Election

- Processes have unique Ids and arranged in a logical ring
- Each process knows its neighbors
  - 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



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

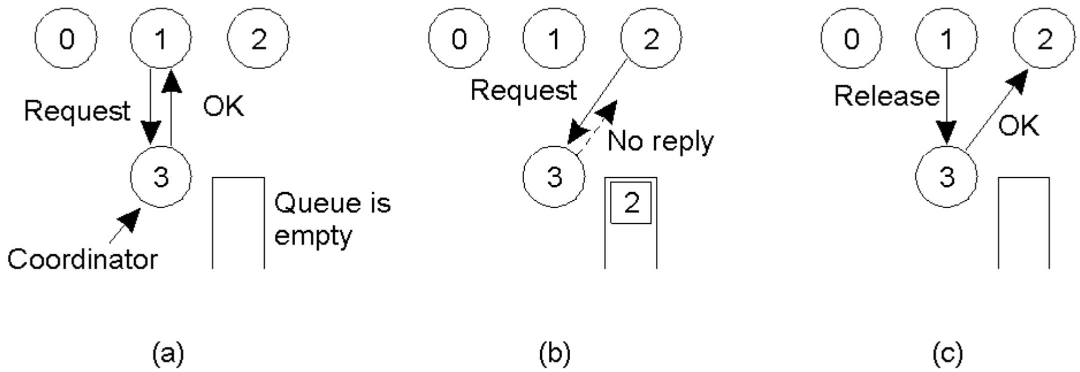
Distributed Synchronization

- 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 grant; if not, queue request
  - Receive *release*: remove next request from queue and send grant

Mutual Exclusion: A Centralized Algorithm



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

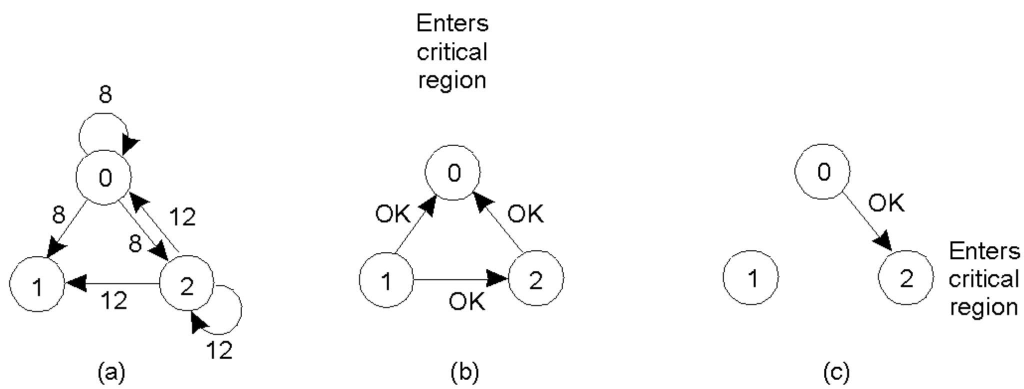
Properties

- 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

- [Ricart and Agrawala]: needs 2(n-1) messages
- Based on event ordering and time stamps
- 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, process *j*
  - Sends *reply* if no contention
  - If already in critical section, does not reply, queue request
  - If wants to enter, compare TS_j with TS_k and send reply if $TS_k<TS_j$, else queue

A Distributed Algorithm

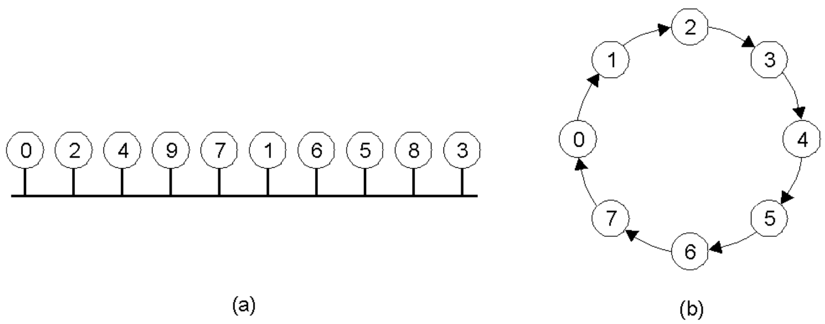


a) Two processes want to enter the same critical region at the same moment.
b) Process 0 has the lowest timestamp, so it wins.
c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

Properties

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

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
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 three mutual exclusion algorithms.

Transactions

- Transactions provide higher level mechanism for *atomicity* of processing in distributed systems
  - Have their origins in databases
- Banking example: Three accounts A:$100, B:$200, C:$300
  - Client 1: transfer $4 from A to B
  - Client 2: transfer $3 from C to B
- Result can be inconsistent unless certain properties are imposed on the accesses

Client 1	Client 2
Read A: $100	
Write A: $96	
	Read C: $300
	Write C:$297
Read B: $200	
	Read B: $200
	Write B:$203
Write B:$204	

ACID Properties

- *Atomic:* all or nothing
- *Consistent*: transaction takes system from one consistent state to another
- *Isolated*: Immediate effects are not visible to other (serializable)
- *Durable:* Changes are permanent once transaction completes (commits)

Client 1	Client 2
Read A: $100	
Write A: $96	
Read B: $200	
Write B:$204	
	Read C: $300
	Write C:$297
	Read B: $204
	Write B:$207