Election & Mutual Exclusion Algorithms

Election Algorithms

- Many Distributed Systems require a *single* process to act as *coordinator* (for various reasons).
  - Time server in the Berkley's algorithm
  - Coordinator in the two-phase commit protocol
  - Master process in distributed computations
  - Master database server
- Coordinator may fail → the distributed group of processes must execute an **election algorithm** to determine a new coordinator process.

Election Algorithms

For simplicity, we assume the following:

- Processes each have a unique, positive identifier.
  - Processor ID
  - IP number
- All processes know all other process identifiers.
- The process with the highest valued identifier is duly elected coordinator.

Goal of Election Algorithms

The overriding goal of all election algorithms is to have all the processes in a group *agree* on a coordinator.

Bully: “the biggest guy in town wins”.

The “Bully” Election Algorithm (1)

Assumes:

- Reliable message delivery (but processes may crash)
- The system is synchronous (timeouts to detect a process failure)
- Each process knows which processes have higher identifiers and can communicate with them

The "Bully" Election Algorithm (2)

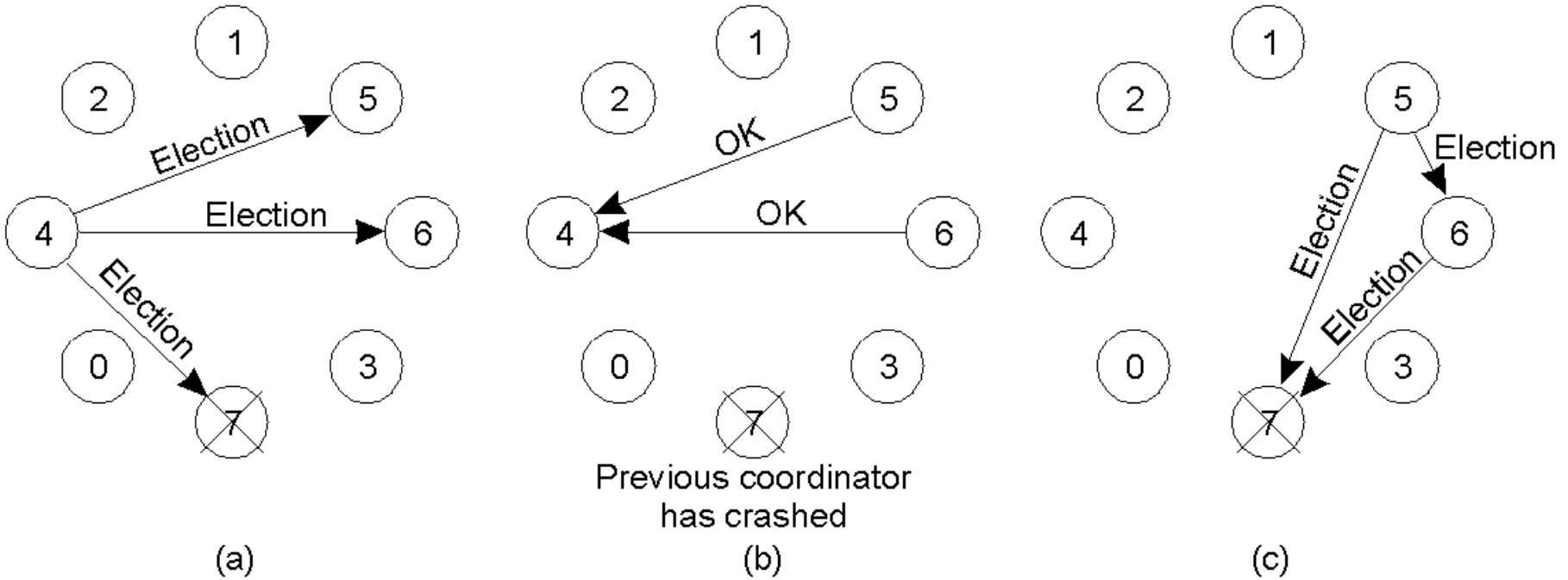
When any process, P, notices that the coordinator is no longer responding it initiates an election:

- P sends an ELECTION message to all processes with higher id numbers.
- If no one responds, P wins the election and becomes coordinator.
- If a higher process responds, it takes over. Process P's job is done.

The “Bully” Election Algorithm (3)

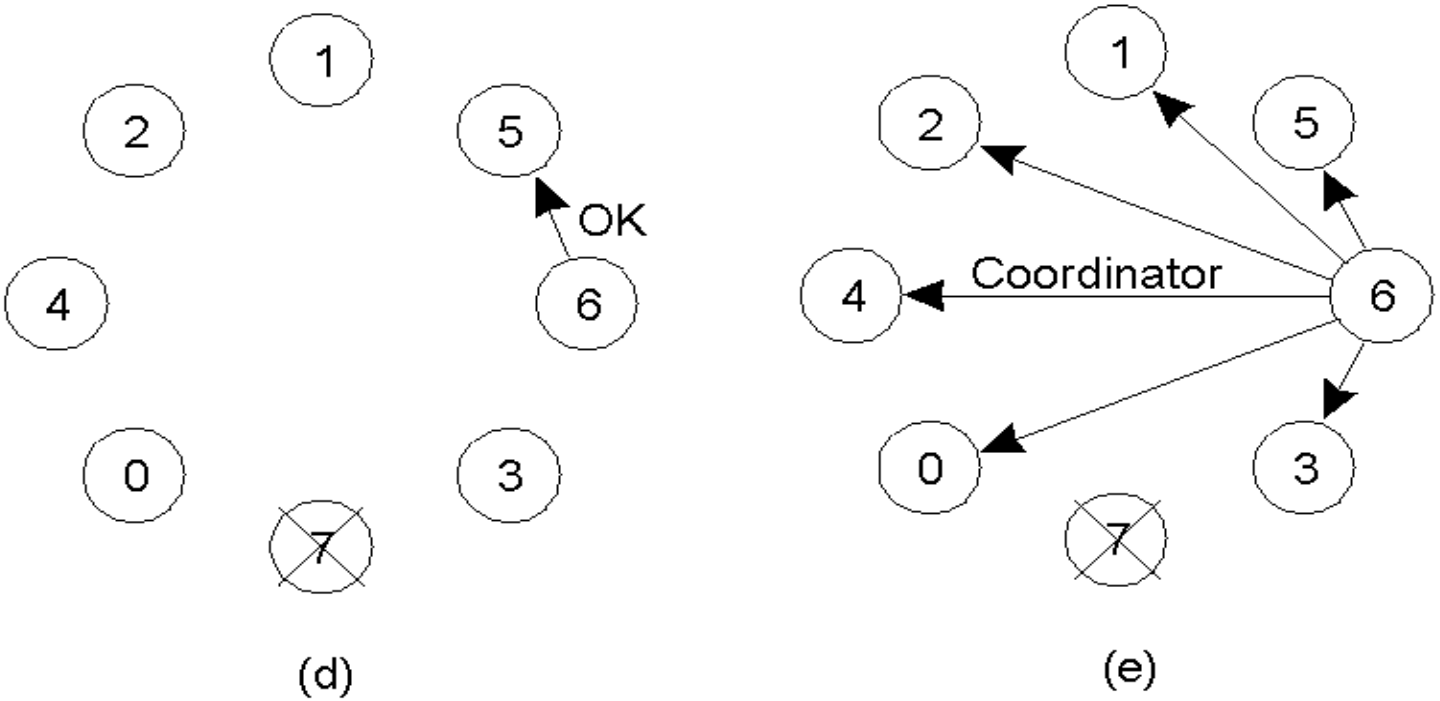
- At any moment, a process can receive an ELECTION message from one of its lower-numbered colleagues.
- The receiver sends an OK back to the sender and conducts its own election.
- Eventually only the bully process remains. The bully announces victory to all processes in the distributed group.

The "Bully" Election Algorithm (4)



- When a process "notices" that the current coordinator is no longer responding (4 deduces that 7 is down), it sends out an ELECTION message to any higher numbered process.
- If none responds, it (ie. 4) becomes the coordinator (sending out a COORDINATOR message to all other processes informing them of this change of coordinator).
- If a higher numbered process responds to the ELECTION message with an OK message, the election is cancelled and the higher-up process starts its own election

The "Bully" Election Algorithm (5)



- 6 wins the election
- When the original coordinator (ie. 7) comes back on-line, it simply sends out a COORDINATOR message, as it is the highest numbered process (and it knows it).
- Simply put: the process with the highest numbered identifier ***bullies*** all others into submission.

Checking your understanding

What happens when two processes detect the demise of the coordinator simultaneously and both decide to hold an election?

Mutual Exclusion

Mutual Exclusion within Distributed Systems

- *n* processes accessing a shared resource
  - File on a hard drive
  - Database entry
  - Printer,…
- It is necessary to protect the shared resource using "mutual exclusion"

Terminology (Uniprocessor)

- *Critical section* is a non-re-entrant piece of code that can only be executed by one process at a time.
- *Mutual exclusion* is a collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions.
- *Semaphore* is a datatype (integer) used to implement mutual exclusion.

- Java synchronized methods:
  - `int synchronized criticalSection(){…}`

Mutual exclusion

Essential requirements for mutual exclusion:

- **Safety**: At most one process may execute in the critical section (CS) at a time
- **Liveness**: Requests to enter and exit CS eventually succeed

Liveness implies freedom of deadlocks and starvation (indefinite postponements of entry for a process that has requested it)

DS Mutual Exclusion: Techniques

Two major approaches:

- **Centralized**: a single coordinator controls whether a process can enter a critical region.
- **Distributed**: the group *confers* to determine whether or not it is safe for a process to enter a critical region.

Centralized Algorithm

Assume a coordinator has been elected.

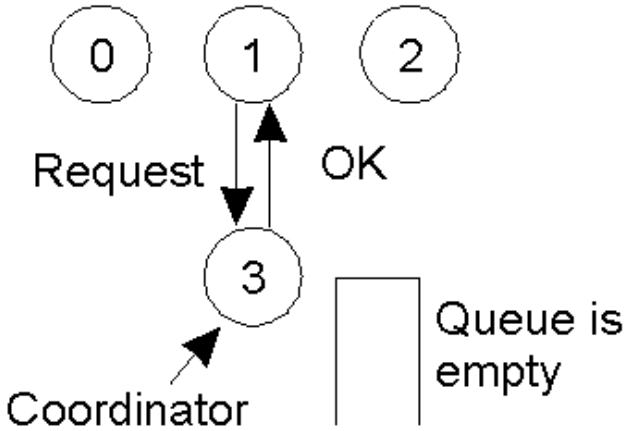
- A process sends a message to the coordinator requesting permission to enter a critical section. If no other process is in the critical section, permission is granted.
- If another process then asks permission to enter the same critical region, the coordinator does not reply (Or, it sends "permission denied") and queues the request.
- When a process exits the critical section, it sends a message to the coordinator.
- The coordinator takes first entry off the queue and sends that process a message granting permission to enter the critical section.

Centralized Algorithm (2)

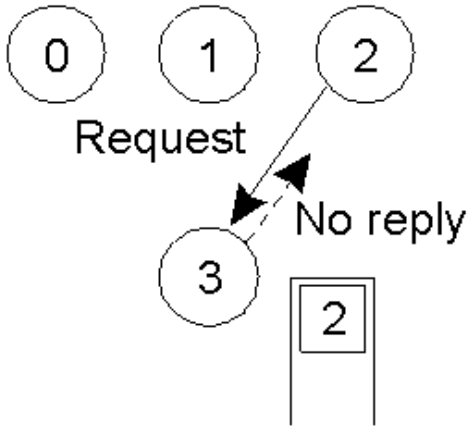
a) Process 1 asks the coordinator for permission to enter a critical region.Permission is granted

b) Process 2 asks for permission to enter the same region. No reply.

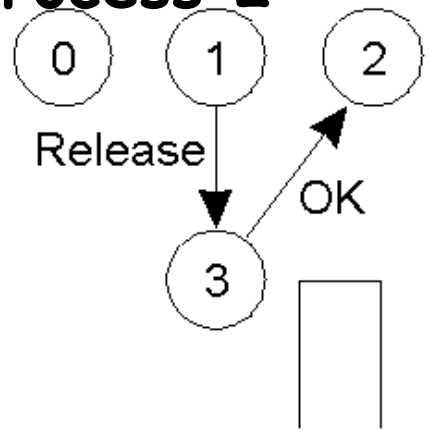
c) When Process 1 quits the critical region, it tells the coordinator, which then replies to Process 2



(a)



(b)



(c)

Comments:

Advantages:

- It works.
- It is fair.
- There's no process starvation.
- Easy to implement.

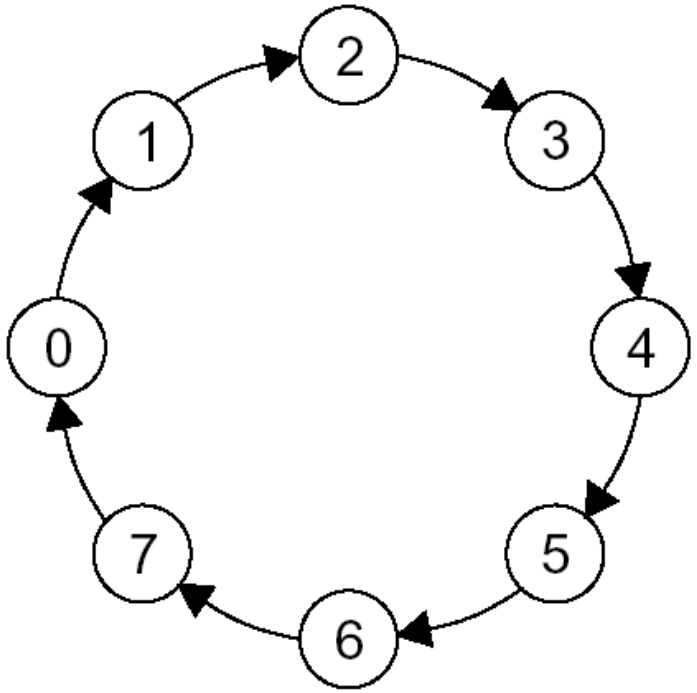
Disadvantages:

- There's a single point of failure!
- The coordinator is a bottleneck on busy systems.

Critical Question: When there is no reply, does this mean that the cordinator is "dead" or just busy?

Token-Ring Algorithm

- Processes are organized into a *logical ring.*
- A *token* circulates around the ring
- A critical region can only be entered when the token is held
  - No token, no access!
- When the critical region is exited, the token is released.



Comments

Advantages:

- It works (as there's only one token, so mutual exclusion is guaranteed).
- It's fair – everyone gets a shot at grabbing the token at some stage.

Disadvantages:

- Lost token! How is the loss detected (it is in use or is it lost)? How is the token regenerated?
- Process failure can cause problems – a broken ring!
- Every process is required to maintain the current logical ring in memory – not easy.

Distributed Algorithm (1)

Ricart and Agrawala algorithm (1981) assumes there is a mechanism for "totally ordering of all events" in the system (e.g. Lamport's algorithm) and a reliable message system.

- A process wanting to enter critical sections (cs) sends a message with (cs name, process id, current time) to all processes (including itself).
- When a process receives a cs request from another process, it reacts based on its current state with respect to the cs requested.

Distributed Algorithm (2)

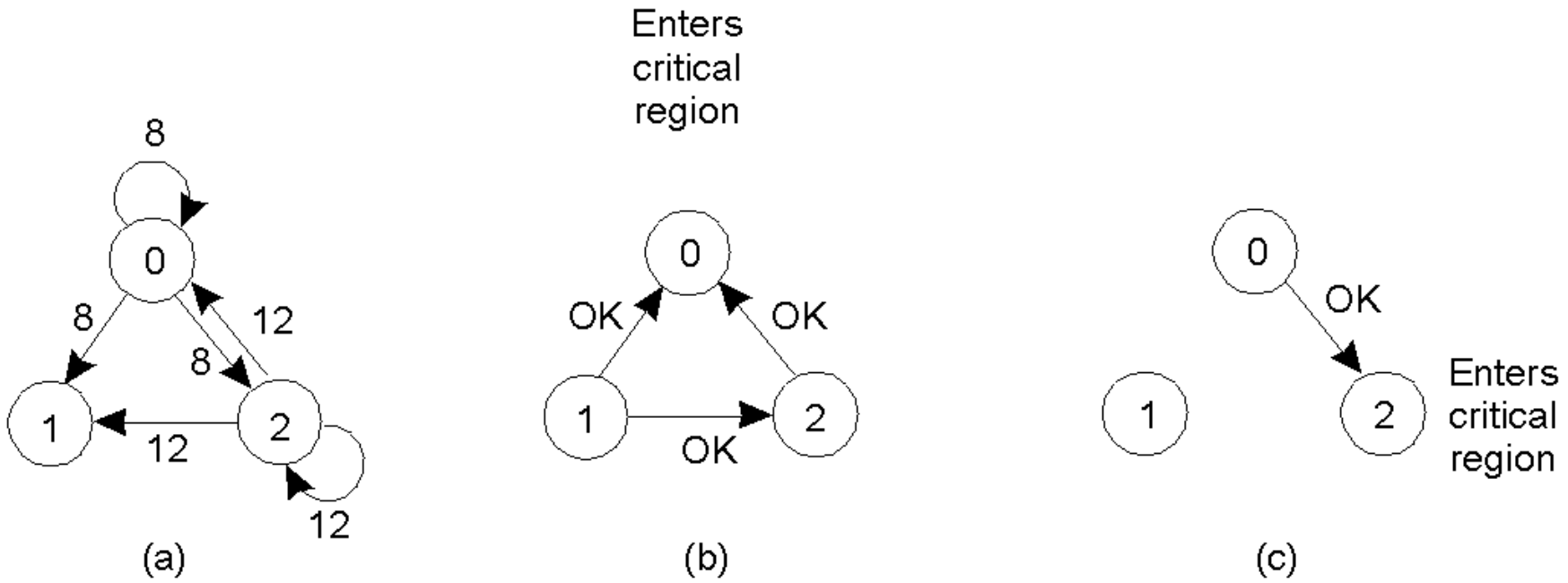
There are three possible cases:

1. If the receiver is not in the cs and it does not want to enter the cs, it sends an OK message to the sender.
2. If the receiver is in the cs, it does not reply and queues the request.
3. If the receiver wants to enter the cs but has not yet, it compares the timestamp of the incoming message with the timestamp of its message sent to everyone. *{The lowest timestamp wins.}*
   - If the incoming timestamp is lower, the receiver sends an OK message to the sender.
   - If its own timestamp is lower, the receiver queues the request and sends nothing.

Distributed Algorithm (3)

- After a process sends out a request to enter a cs, it waits for an OK from all the other processes. When all are received, it enters the cs.
- Upon exiting cs, it sends OK messages to all processes on its queue for that cs and deletes them from the queue.

The Distributed Algorithm in Action



Process 0 and 2 wish to enter the critical region “at the same time”.
Process 0 wins as it’s timestamp is lower than that of process 2.
When process 0 leaves the critical region, it sends an OK to 2.

Comments

Advantages:

- It works.
  - The algorithm works because in the case of a conflict, the lowest timestamp wins as everyone agrees on the total ordering of the events in the distributed system.
- There is no single point of failure

Disadvantages:

- We now have multiple points of failure!!!
- A "crash" is interpreted as a *denial of entry* to a critical region.
- (A patch to the algorithm requires all messages to be ACKed).
- Worse is that all processes must maintain a list of the current processes in the group (and this can be tricky)
- Worse still is that one overworked process in the system can become a *bottleneck* to the entire system – so, everyone slows down.

Comparison: Mutual Exclusion Algorithms

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

None are perfect – they all have their problems!

The "Centralized" algorithm is simple and efficient, but suffers from a single point-of-failure.

The "Distributed" algorithm has nothing going for it – it is slow, complicated, inefficient of network bandwidth, and not very robust. It "sucks"!

The "Token-Ring" algorithm suffers from the fact that it can sometimes take a long time to reenter a critical region having just exited it.

All perform poorly when a process crashes, and they are all generally poorer technologies than their non-distributed counterparts. Only in situations where crashes are very infrequent should any of these techniques be considered.