**Distributed Coordination**

**Centralized Mutual Exclusion**

- **Central Server Approach**
  - All processes contact central server to obtain permission to enter CS
  - Pros: Simple to implement
  - Cons: Can be slow (time to transmit release and grant messages); central server is bottleneck

- **Ring-Based Approach**
  - Arrange processes in logical ring
  - Each process has communication channel to the next process
  - Pass “token” around ring; token grants access to CS
  - Pros: Simple, no central bottleneck
  - Cons: Potentially large delay; wastes bandwidth

- **Multicast & Logical Clocks**
  - Ricart and Agrawala developed approach based on multicast and Lamport clocks
  - Multicast request for access to other processes; wait for reply
  - Logical timestamps make sure happened-before requirement is met
  - Pros: Short delay (compared to ring)
  - Cons: Consumes lots of bandwidth

- **Maekawa’s Voting Algorithm**
  - Not necessary for all processes to grant access, only need subset of all processes
  - Each process maintains a “voting set”
  - All voting sets are the same size
  - Make sure subsets used by any two processes overlap
  - For all voting sets, \( V_i \cap V_j \neq \emptyset \)
  - Pros: Requires less bandwidth than previous approach
  - Cons: Can cause deadlock! How?

**Deadlock Example**

- Seven processes, seven voting sets
- From \( V_0 = \{0, 1, 2\} \), 0, 2 send ack to 0, but 1 sends ack to 1;
- From \( V_1 = \{1, 3, 5\} \), 1, 3 send ack to 1, but 5 sends ack to 2;
- For all voting sets, \( V_2 = \{2, 4, 5\} \), 4, 5 send ack to 2, but 2 sends ack to 0;
- Now, 0 waits for 1, 1 waits for 2, and 2 waits for 0.
- So deadlock is possible!

(To correct this, requests are accepted in happened-before order.)

**Questions**

- What about fault tolerance?
- What happens when messages are lost?
- What happens when a process crashes?
Token Ring Approach

Multicast Approach (Ricart and Agrawala)

“Request (41)”

“Request (34)”
Voting Approach (Maekawa’s Algorithm)

Voting Set Example

\[ V_0 = \{0, 1, 2\} \]
\[ V_1 = \{1, 3, 5\} \]
\[ V_2 = \{2, 4, 5\} \]
\[ V_3 = \{0, 3, 4\} \]
\[ V_4 = \{1, 4, 6\} \]
\[ V_5 = \{0, 5, 6\} \]
\[ V_6 = \{2, 3, 6\} \]
Elections

Ring-Based Election
Bully Algorithm

Bully Algorithm Details

- Any process $P$ can initiate an election.
- $P$ sends Election messages to all processes with higher IDs and awaits OK messages.
- If no OK messages, $P$ becomes the coordinator and sends I won messages to all processes with lower IDs.
- If it receives an OK, it drops out and waits for an I won message.
- If a process receives an Election message, it returns an OK and starts an election.
- If a process receives an I won message, it treats the sender as the coordinator.

Bully Algorithm Example

- Process 4 holds an election.
- Process 5 and 6 respond, telling 4 to stop.
- Now 5 and 6 each hold an election.
- Process 6 tells 5 to stop.
- 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.
- Select the process with the highest ID.
- Begin an election if just recovered or the coordinator has failed.
- Send Election to the closest downstream node that is alive.
- Sequentially poll each successor until a live node is found.
- Each process tags its ID on the message.
- The initiator picks the node with the highest ID and sends a coordinator message.
- Multiple elections can be in progress. Wastes network bandwidth but does no harm.