CSC2/458 Parallel and Distributed Systems
Mutual Exclusion and Leader Elections

Sreepathi Pai

March 29, 2018

URCS

Outline

Mutual Exclusion Using Voting

Misra's Token Recovery Algorithm

Election Algorithms

Outline

From the previous lecture

- Does a process need to wait for all replicas to reply before checking majority?
  - No [it would NOT (thanks, Mohsen!) solve the problem raised by Andrew, but would lead to lower utilization]
- How many processes need to fail?
  - $f >= m - N/2$, where
  - $m = N/2 + 1$
- Does this mean mutual exclusion can be violated?
  - Yes (with very low probability, see Lin et al. 2014)

Different Types of Failures (Thomas)

- How does fail recovery compare with fail stop?
  - Fail stop: Process operates correctly, fails in a detectable way and remains failed
  - Fail recovery: Process fails and "restarts"

Outline

Recall Token-based Mutual Exclusion

- A token circulates in an (unidirectional) ring
  - Process i sends token to Process $i + 1$ (modulo N)
- A process holding the token can perform actions on shared resources
  - i.e. it is in the critical section
- A tokens can be lost
  - released by process i but not received by process j

Loss of token

- Two problems
  - Detecting loss
  - Regenerating a single token

One possible solution

- Detect loss of token using *timeouts*
- Perform leader election
  - Leader generates new token
- This solution in a few slides

Misra's algorithm for detecting token loss and regeneration

- Use two tokens X and Y
  - X is also the mutual exclusion token (but not Y)
  - X and Y detect the loss of each other
- Assume in order receipt

Key Insight

"A token at a process p_i can guarantee the other token is lost if since this token's last visit to p_i, neither this token nor p_i have seen the other token."

- Misra, 1983, Detecting Termination of Distributed Computations Using Markers, PODC

- What does it mean for:
  - a process to have seen a token?
  - for a token to have seen the other token?

The Algorithm: Setup

- Associate nX and nY, two integers with X and Y
- Initialize nX and nY to $+1$ and -1 respectively
- Each token carries its value with it (i.e nX or nY)
- Each process p_i contains a m_i initialized to zero
    - remembers the last token seen and its value

The Algorithm: Working

When tokens encounter each other:

```
nX = nX + 1
nY = nY - 1
```

When p_i encounters Y (analogous code to encountering X not shown):

```
if m_i == nY: /* token X is lost */
    /* regenerate token X */
    nY -= 1
    nX = -nY
else:
    m_i = nY
end if
```

Do we need infinite precision?

- nX can become arbitrarily large
- nY can become arbitrarily small
- Can we avoid this?
  - What is the invariant we need to maintain?
  - When are counters updated?
  - How many such events can happen between two visits to p_i?

Other notes

Misra proposed this algorithm for *termination* detection. We will revisit it.

But can you see how it may apply?

- All processes are in either IDLE or ACTIVE
- Receiving a message marks process as ACTIVE
- Processes can only quit when all of them are IDLE and there are no messages in flight

Outline

Electing Leaders

- Initiating an election
  - Anytime
- Detecting a winner and making sure everybody agrees on the same winner
  - Using process IDs to break ties for example

Ring-based Elections: Selective Extension

- (Logical) Unidirectional ring topology
- Two message types, both contain a process ID:
  - ELECTION
  - ELECTED

Algorithm: Part I

A process can initiate an election *anytime*. Process p_i does this by sending a ELECTION(p_i) to its neighbour and "marking itself" as participating in an election.

On receiving message ELECTION(X), a process p_j:

```
if X > p_j:
   participating = T
   send(ELECTION(X))
elif X < p_j:
   participating = T
   send(ELECTION(p_j))
elif X == p_j:
   send(ELECTED(p_j))
```

Algorithm: Part II

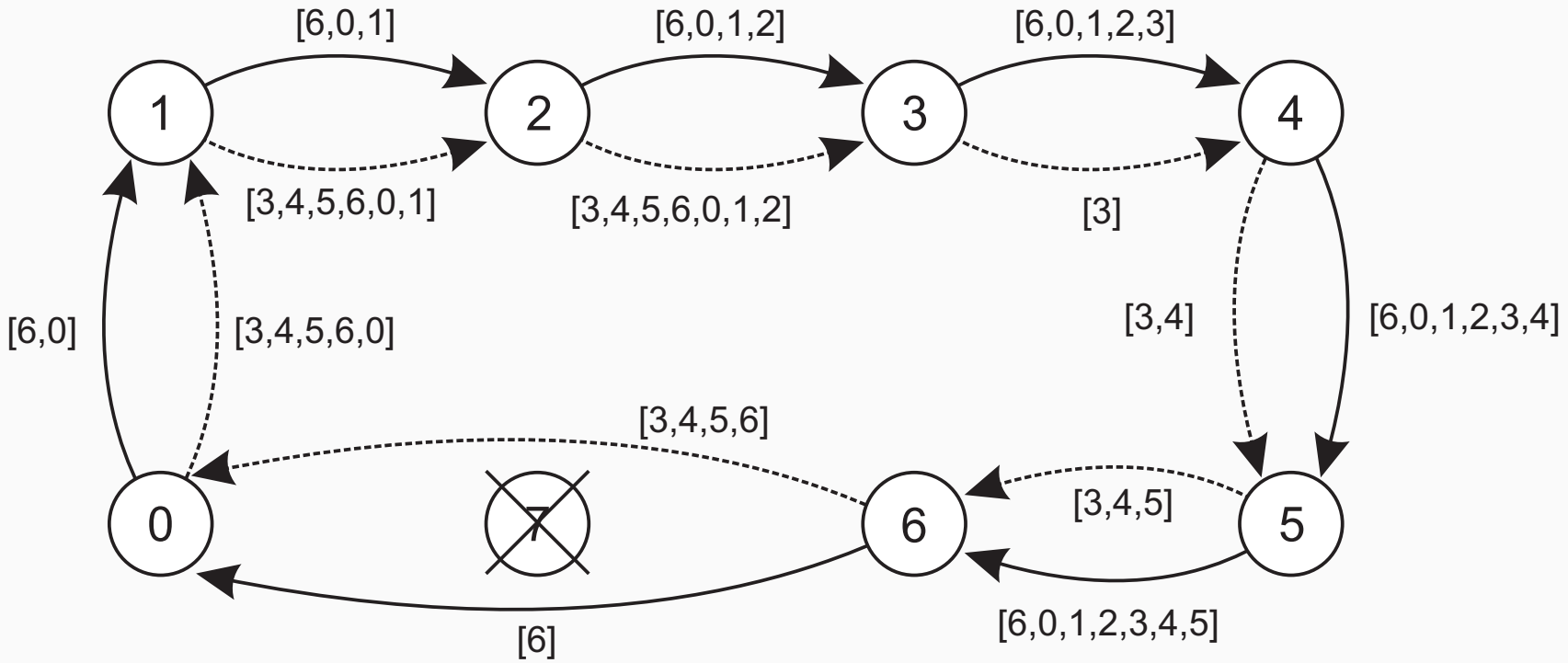
When receiving ELECTED(Y):

```
participating = F
coordinator = Y

if Y != p_j:
   send(ELECTED(Y))
```

Textbook has slight modifications

- Sends lists instead of one number
- Skips dead nodes

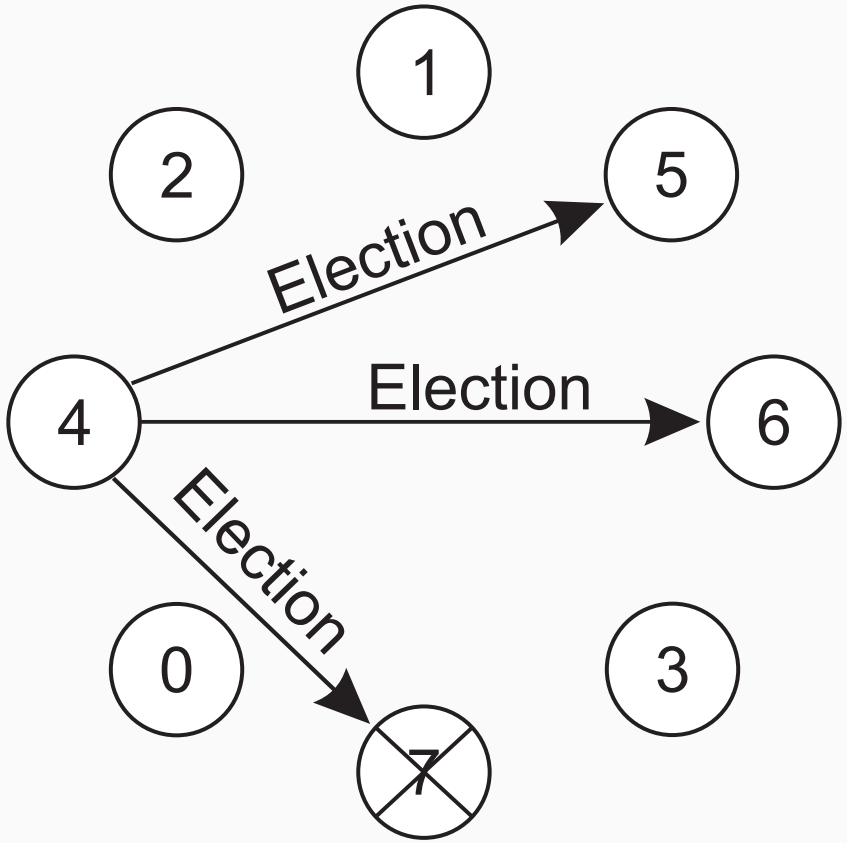


The Bully Algorithm

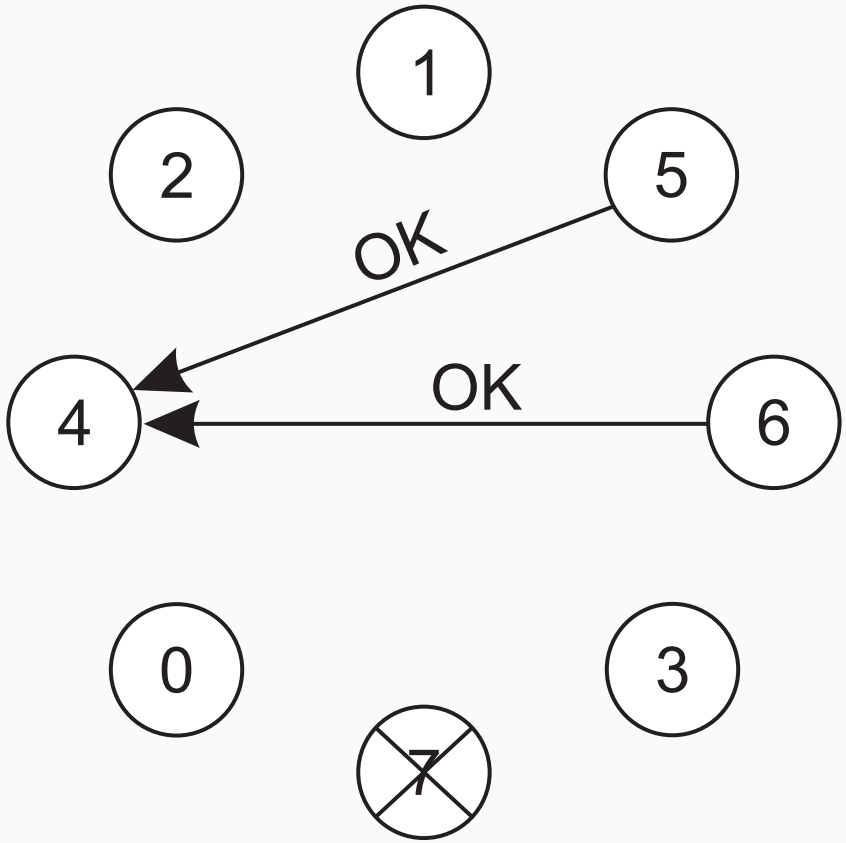
The coordinator with the highest process ID always wins.

- Three types of messages:
  - ELECTION (initiation)
  - OK (resolution)
  - COORDINATOR (verdict)

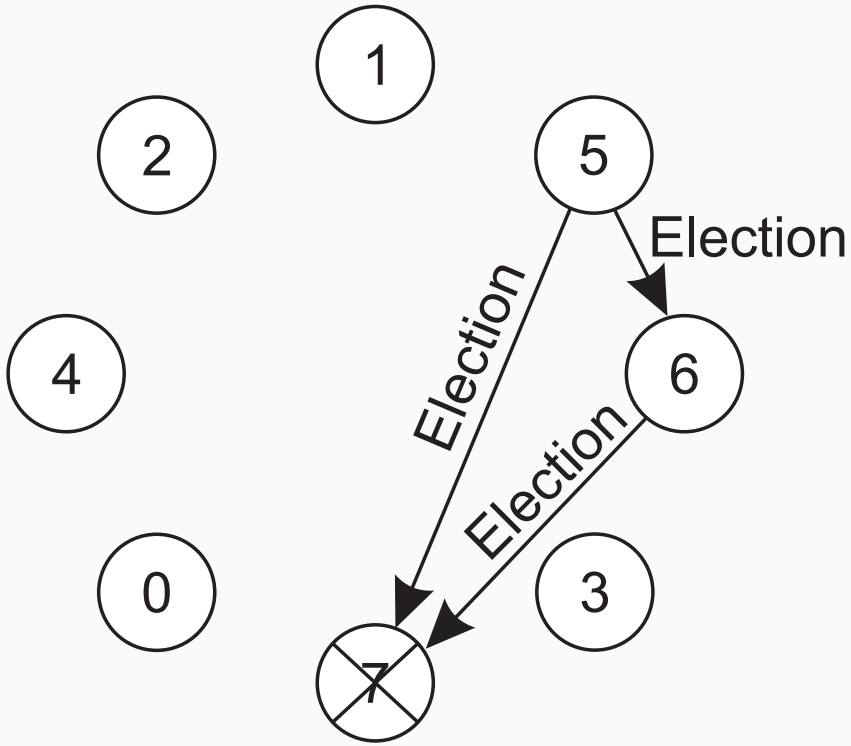
Bully Algorithm in Action: Initiation



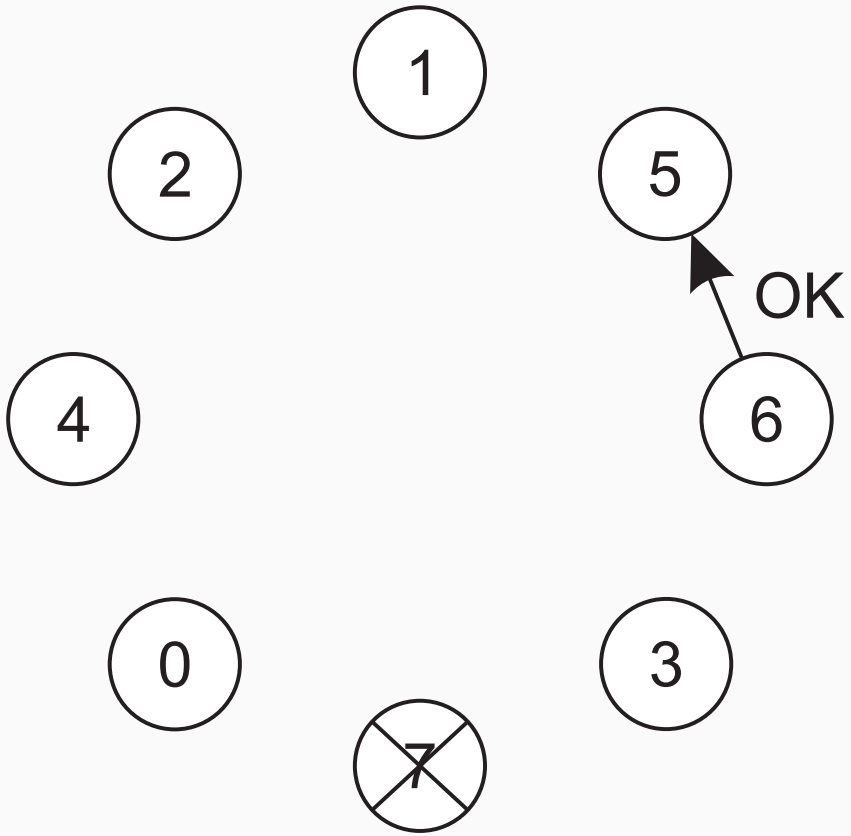
Bully Algorithm in Action: Resolution



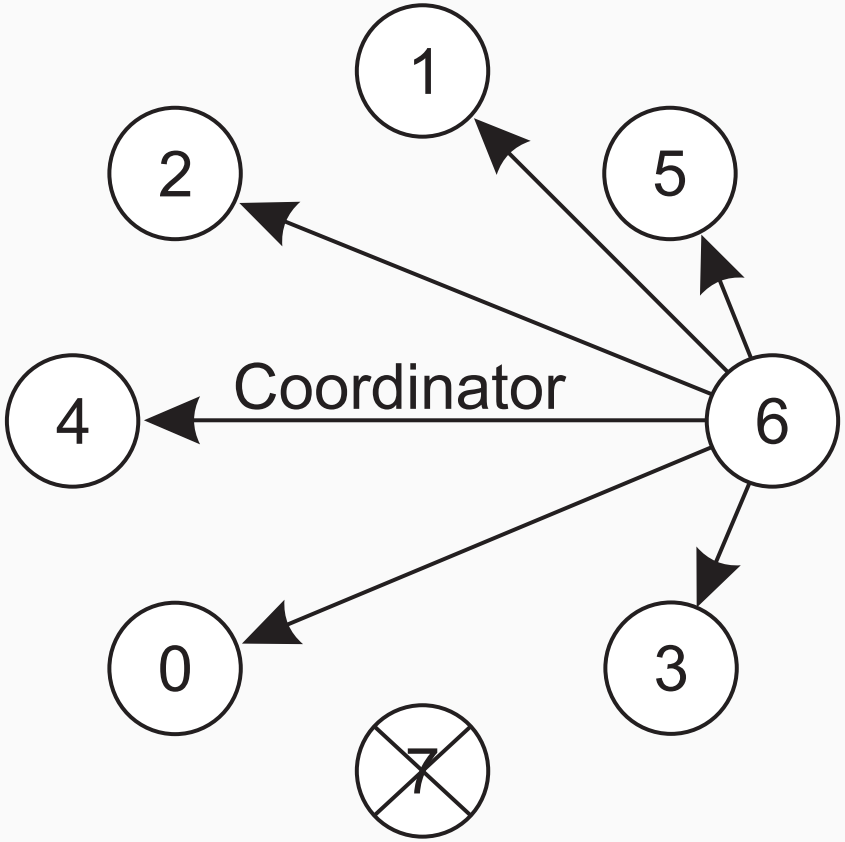
Bully Algorithm in Action: Further Elections



Bully Algorithm in Action: Resolution



Bully Algorithm in Action: Final Verdict



Algorithm

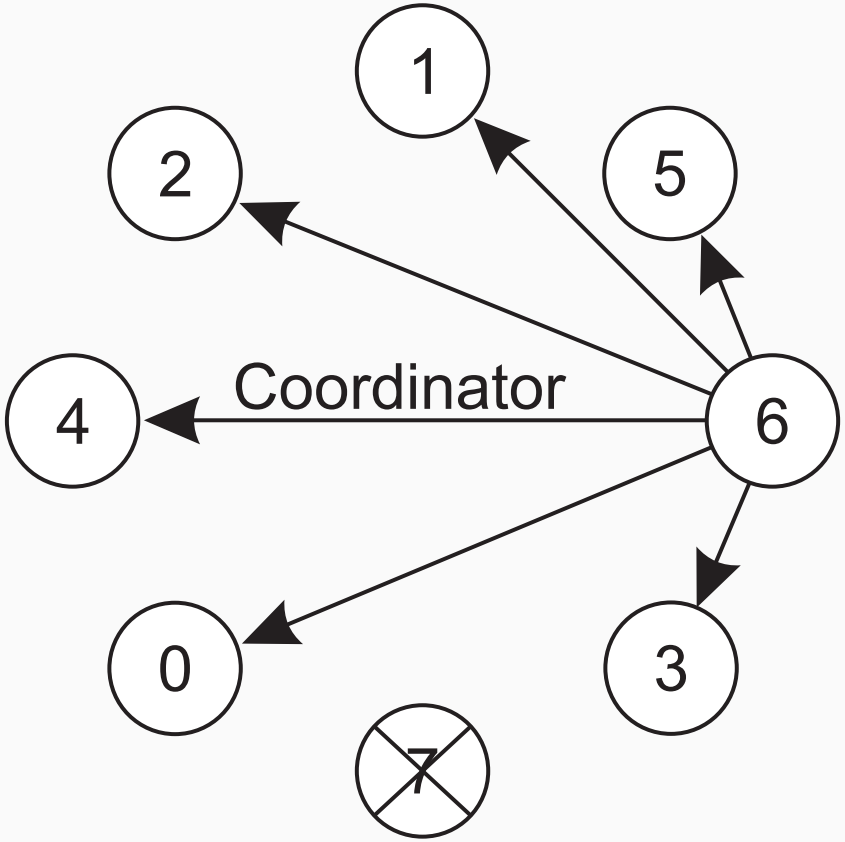
Any process p_i can initiate an election at any time:

- Send ELECTION message to all processes p_k such that $k > i$
- Wait for OK replies
- If no replies (within a timeout), process p_i has won and announces win using COORDINATOR

On receiving an ELECTION message:

- Send OK to sender
  - Sender cannot become a coordinator
- Initiate election if any higher processes known to exist
  - if not, process is new coordinator, send COORDINATOR

What happens when 7 comes back online?



Interesting Extensions

- Wireless networks
  - Small, dynamic, no fixed topology
- P2P networks
  - Large, dynamic, may need multiple coordinators
- See textbook for details
  - Will revisit some of these topics on a P2P lecture

Acknowledgements

All figures from van Steen and Tanenbaum, 3rd Edition.