Leader Election

CS425 /ECE428 – DISTRIBUTED SYSTEMS – FALL 2021

Material derived from slides by I. Gupta, M. Harandi,
J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya

Leader Election

Centralized algorithms are simple

- E.g., sequencer
- E.g., mutual exclusion

How to choose “leader”

- … at start-up time?
- … if a leader fails?

What is Election?

In a group of processes, elect a Leader to undertake special tasks.

What happens when a leader fails (crashes)

- Some process detects this
- Then what?

Focus of this lecture: Election algorithm

- 1. Elect one leader (only among the non-faulty processes)
- 2. All non-faulty processes agree on who is the leader

Problem

Any process can call for an election.

A process can call for at most one election at a time.

Multiple processes can call an election simultaneously.

- *All of them together must yield a single leader only*
- *The result of an election should not depend on which process calls for it.*

Problem Specification

L(*p*) – process that *p* believes to be the leader

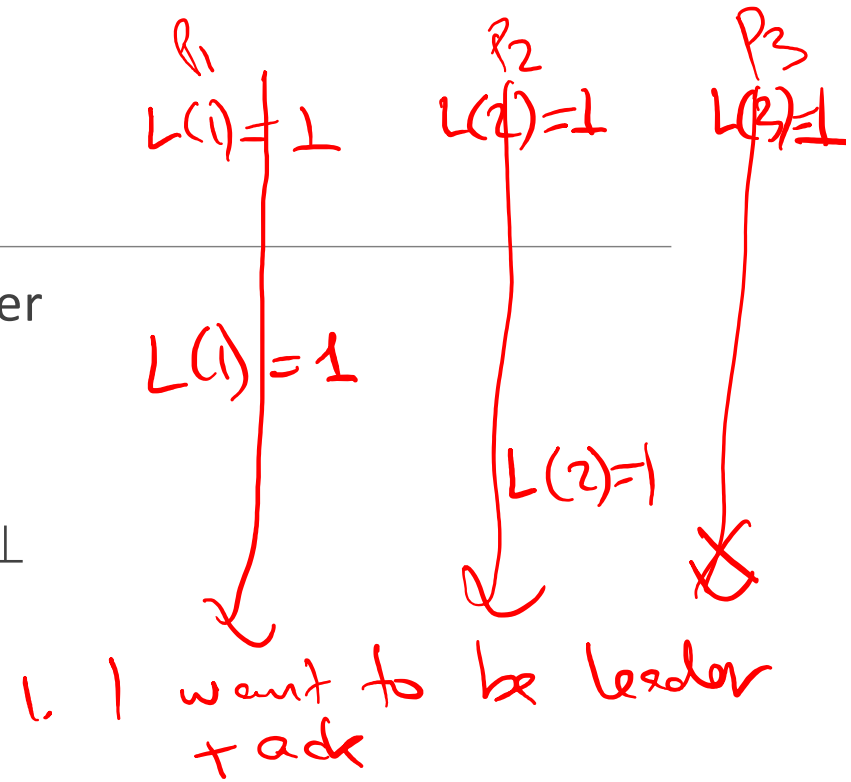
- L(p) = ⊥ — no leader yet

Safety requirement

- ∀non-faulty p, p': If L(p) != ⊥ and L(p') != ⊥
  - L(p) = L(p')
  - L(p) is non-faulty

Liveness requirement

- ∀non-faulty p: eventually L(p) != ⊥



Ring networks

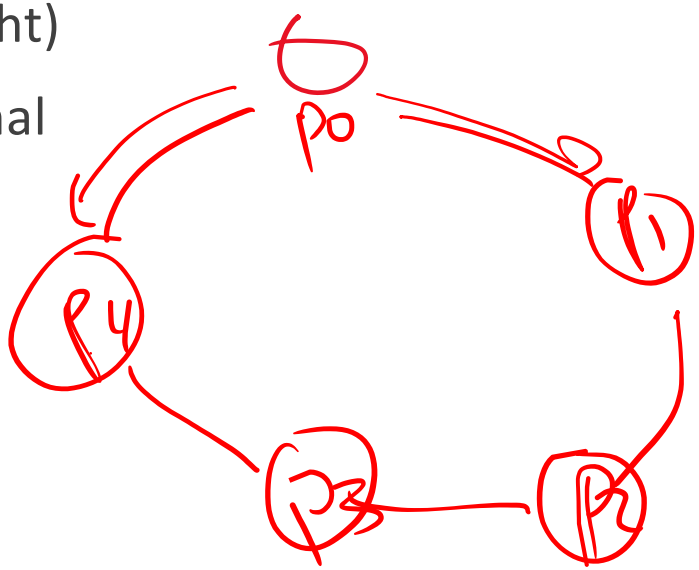
Each process has two neighbors (left & right)

Communication can be uni- or bi-directional

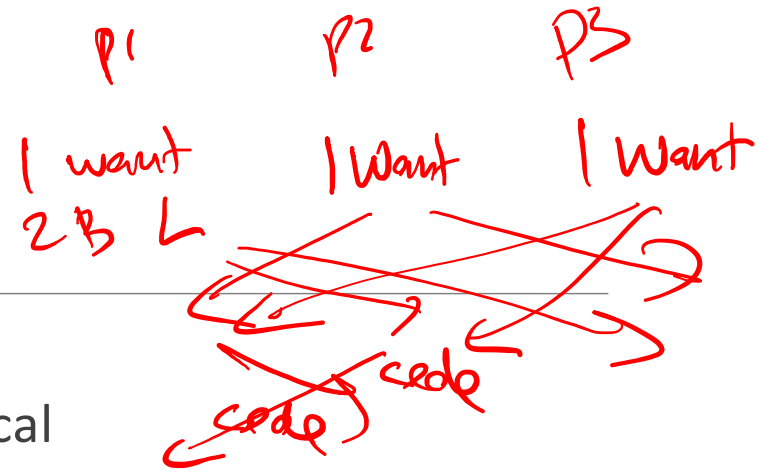
No failures

Why ring?

- Not representative of current systems
- Easy to analyze



Symmetry



Anonymous processes: no unique identifier

- Each process's initial state is precisely identical

Theorem: no anonymous leader election possible

- Each process starts in *same* state
- Each round, a process receives the *same* message from others (symmetry)
- Either all processes think they're the leader (not safe) or none of them do (not live)

Breaking Symmetry

Theorem shows need for unique IDs

- Even in synchronous systems
- Even if number of processes known

Each process *p* has *p.id*

- p != p’ => p.id != p’.id
- Usually ensure that L(p).id >= p.id

What to use for IDs?

- Serial numbers
- Desirable attributes (bandwidth, CPU load, etc.)
  - Must be careful to avoid duplicates

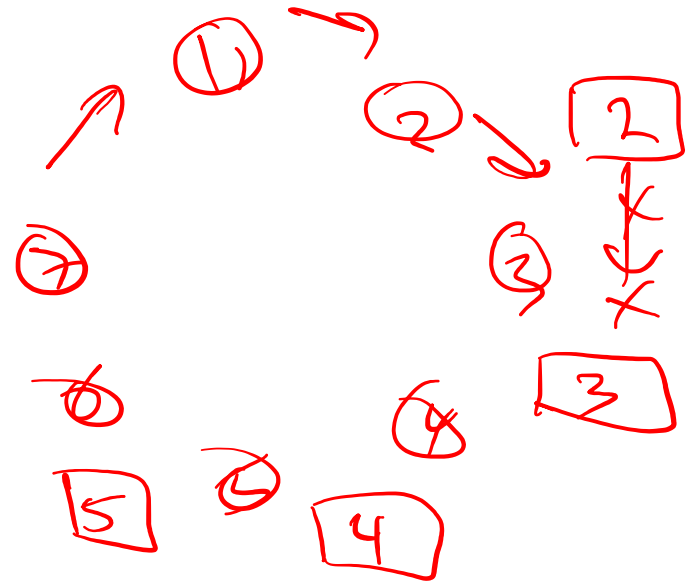
Algorithm 1: (uni-directional) Ring Election [Chang & Roberts'79]

To start election

- Send "election" message with my ID

When receiving message ("election", *id*)

- If *id* > my ID: forward message
  - Set state to "participating"
- If *id* < my ID: send ("election", my ID)
  - Skip if already "participating"
  - Set state to "participating"
- If *id* = my ID: I am elected (why?) send "elected" message
  - "elected" message forwarded until it reaches leader

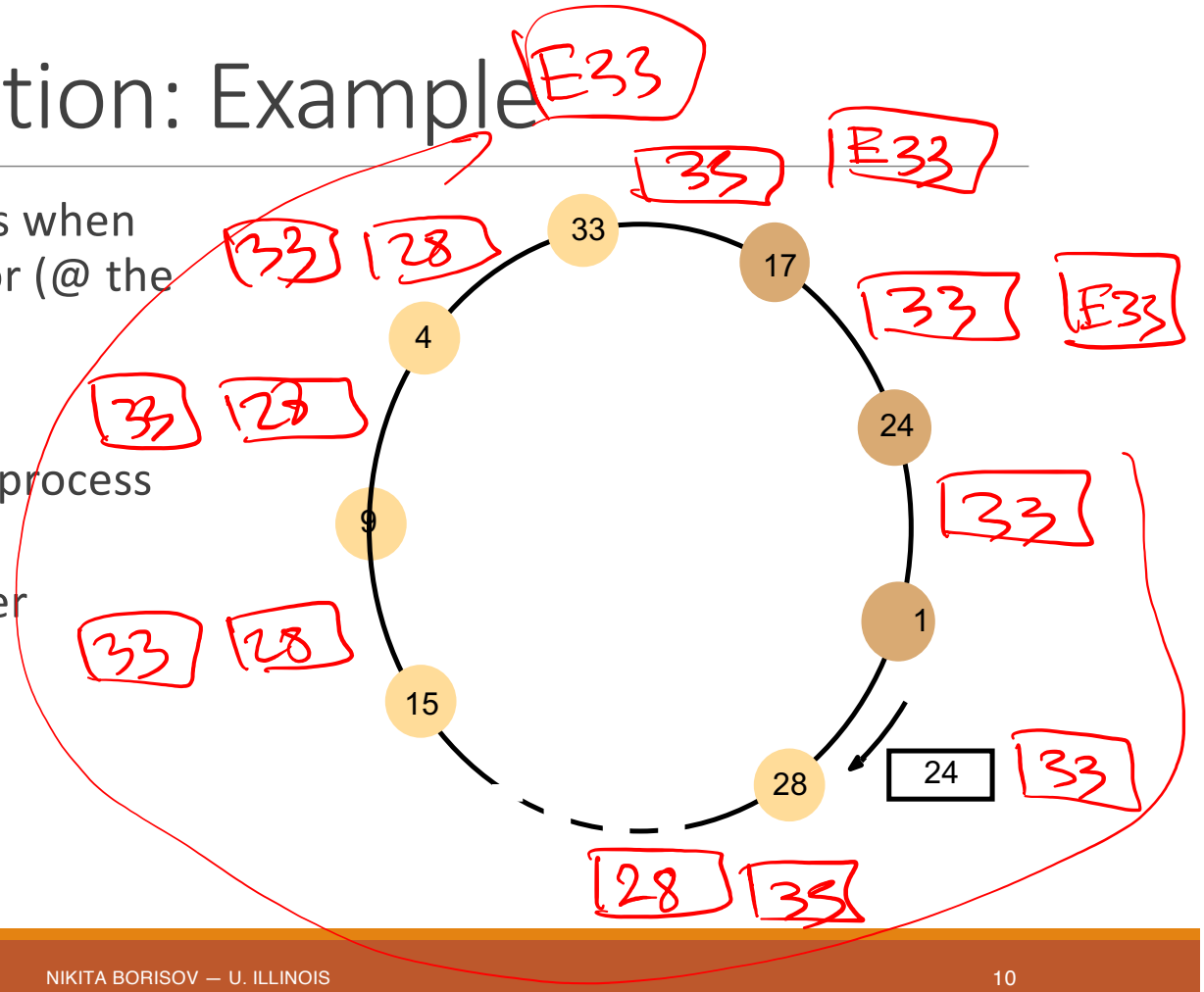


Ring-Based Election: Example

The worst-case scenario occurs when the counter-clockwise neighbor (@ the initiator) has the highest attr.

In the example:

- The election was started by process 17.
- The highest process identifier encountered so far is 24
- (final leader will be 33)



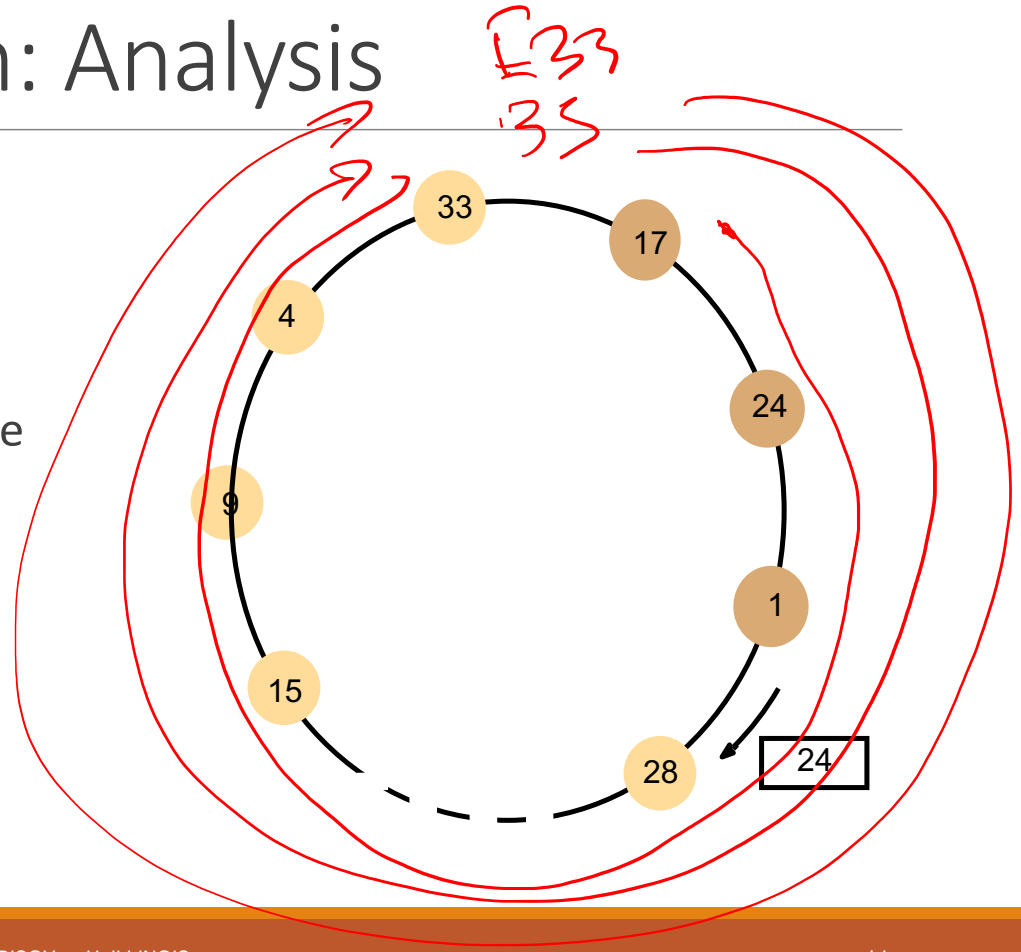
Ring-Based Election: Analysis

In a ring of N processes, in the worst case:

- N-1 election messages to reach the new coordinator
- Another N election messages before coordinator decides it's elected
- Another N elected messages to announce winner

Complexity: $O(N^2)$

- If everyone starts election

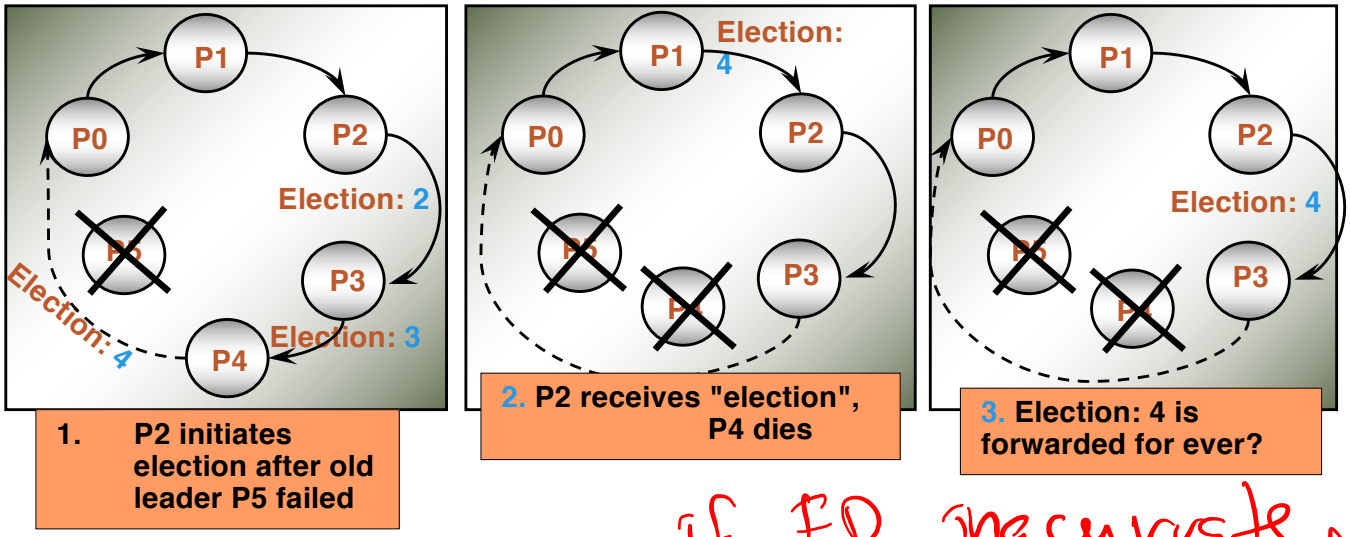


Correctness?

Safety: highest process elected

Liveness: complete after 3N-1 messages

Add Failures



Assumptions

- Failures are detected
- Ring gets repaired

if FD inaccurate
→ unsafe, exclude process from ring

if FD incomplete
→ never detect failure

Algorithm 2: Modified Ring Election

election message tracks *all* IDs of nodes that forwarded it, not just the highest

- Each node appends its ID to the list

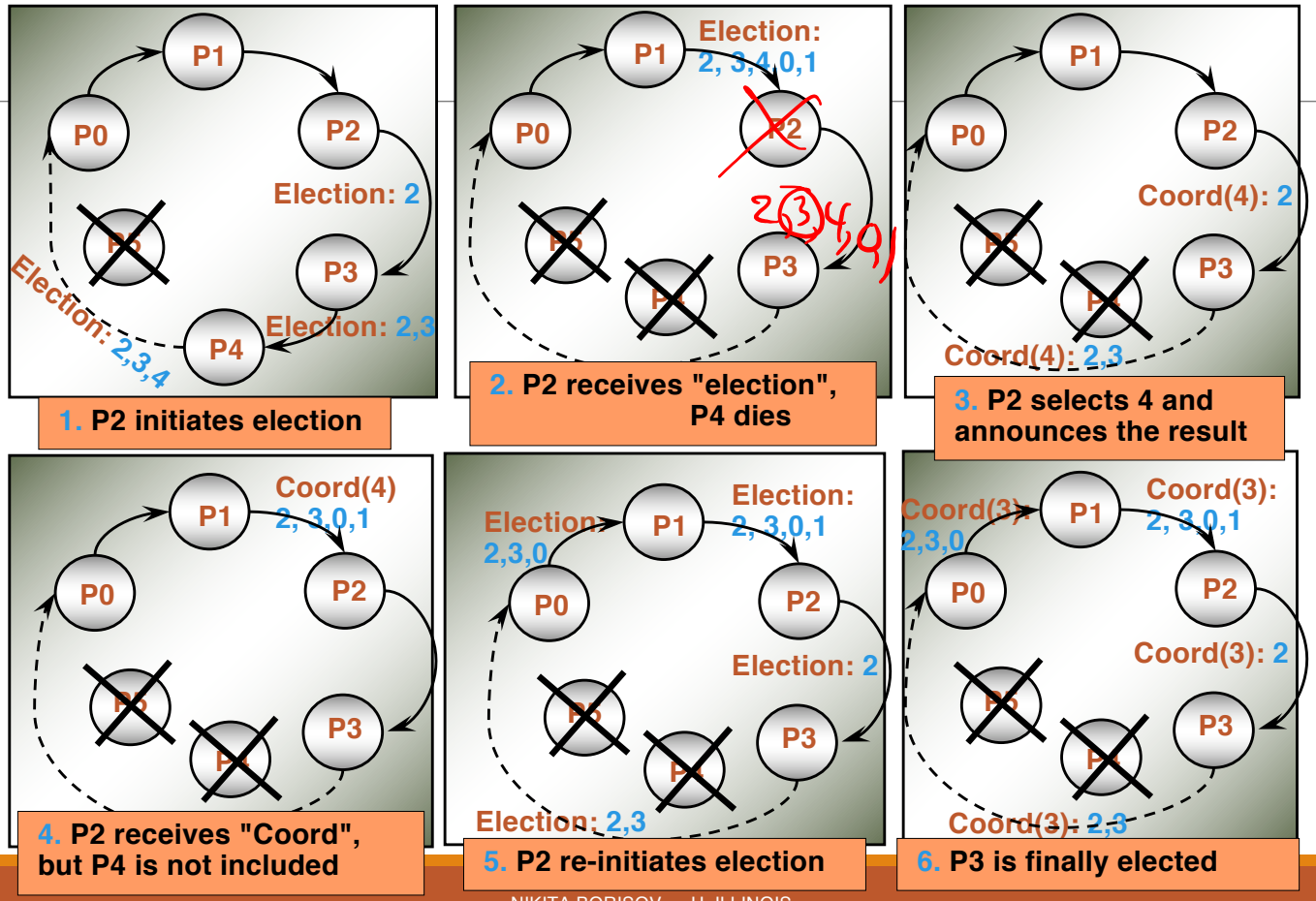
Once message goes all the way around a circle, new coordinator message is sent out

- Coordinator chosen by highest ID in election message
- Each node appends its own ID to coordinator message

When coordinator message returns to initiator

- Election a success if coordinator among ID list
- Otherwise, start election anew

Example: Ring Election



Modified Ring Election

How many messages?

- 2N

Is this better than original ring protocol?

- Messages are larger

Reconfiguration of ring upon failures

- Can be done if all processes "know" about all other processes in the system

What if initiator fails?

- Successor notices a message that went all the way around (how?)
- Starts new election

What if two people initiate at once

- Discard initiators with lower IDs

Asynchronous systems

Can we have a totally correct election algorithm in a fully asynchronous system (no bounds)

- No! Election can solve consensus (see later)

Where might you run into problems with the modified ring algorithm?

- Detect leader failures
- Ring reorganization

Real-world Elections

Synchronous design

- Assume node has failed after a timeout
- Only probabilistically correct

Any-to-any communication

- Set of all potentially correct nodes known

Multicast Algorithm

Start an election

- Multicast <election, my ID> to all processes
- If receive <disagree> from any process
  - Give up election
- If receive <agree> from all processes, then elected
  - Multicast <coordinator, my ID>

Receive <election, *ID*> from process *p*

- If ID > my ID
  - Send <agree> to *p* (unicast)
- If ID < my ID
  - Send <disagree> to *p*
  - Start election (if not already running)

What about failures?

Multicast Algorithm

Start an election

- Multicast <election, my ID> to all processes
- If receive <disagree> from any process
  - Give up election
- If receive <agree> from all processes or timeout, then elected
  - Multicast <coordinator, my ID>

Receive <election, *ID*> from process *p*

- If ID > my ID
  - Send <agree> to *p* (unicast)
- If ID < my ID
  - Send <disagree> to *p*
  - Start election (if not already running)

Can we improve on this?

Multicast Algorithm

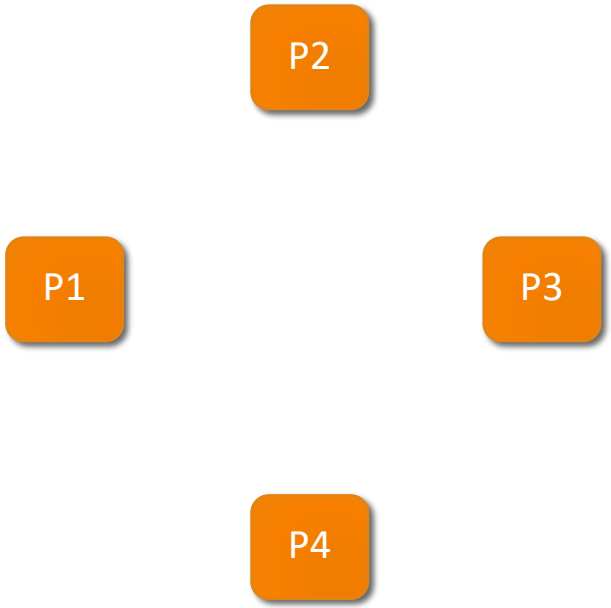
Start an election

- Multicast <election, my ID> to all processes
- If receive <disagree> from any process
  - Give up election
- If ~~receive <agree> from all processes or~~ timeout, then elected
  - Multicast <coordinator, my ID>

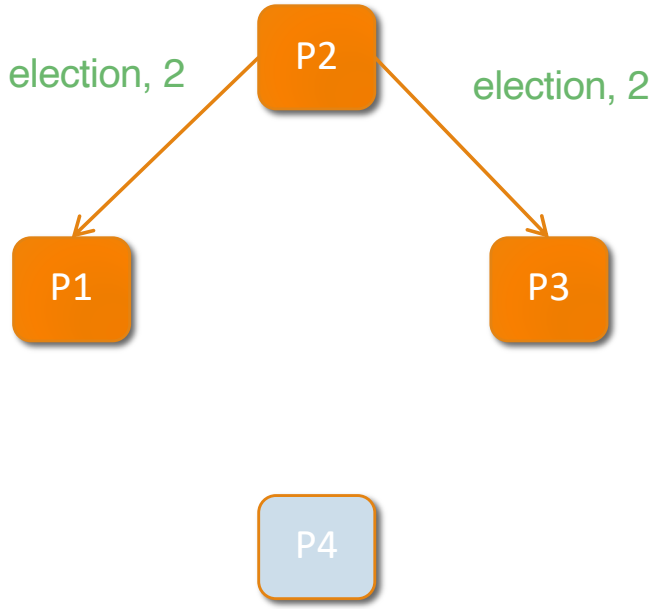
Receive <election, *ID*> from process *p*

- ~~If ID > my ID~~
  - ~~Send <agree> to *p* (unicast)~~
- If ID < my ID
  - Send <disagree> to *p*
  - Start election (if not already running)

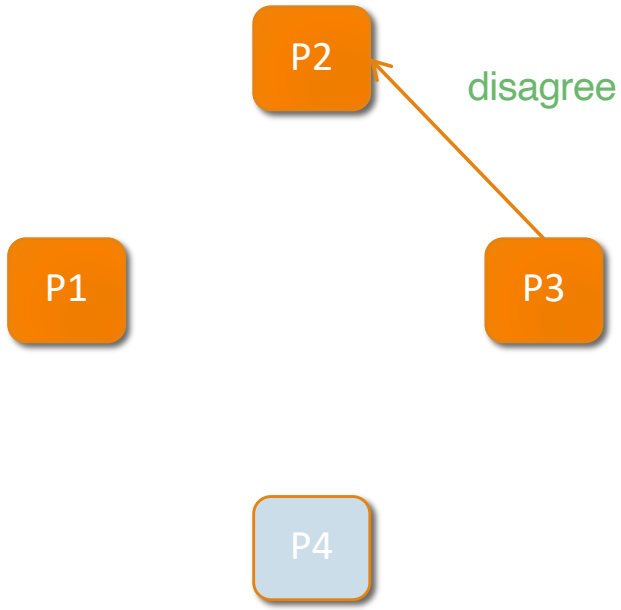
Cascading Failures



Cascading Failures



Cascading Failures



Multicast Algorithm

Start an election

- Multicast <election, my ID> to all processes
- If receive <disagree> from any process
  - Wait for coordinator message
  - Restart election on timeout
- If **timeout**, then elected
  - Multicast <coordinator, my ID>

Receive <election, *ID*> from process *p*

- If ID < my ID
  - Send <disagree> to *p*
  - Start election (if not already running)

Bully Algorithm

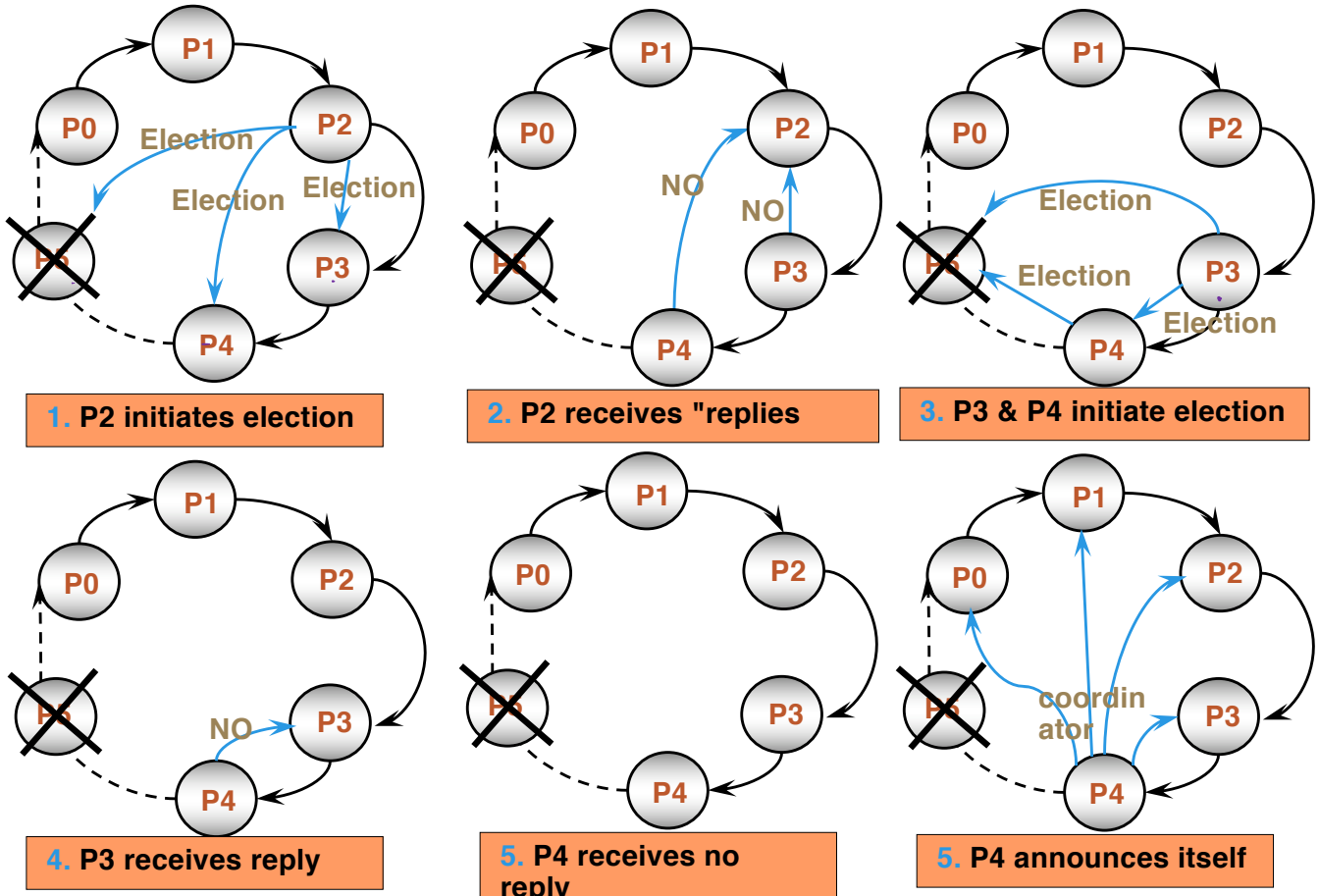
Start an election

- Send <election~~, my ID~~> to processes with higher ID
- If receive <disagree> from any process
  - Wait for coordinator message
  - Restart election on timeout
- If timeout, then elected
  - Multicast <coordinator, my ID>

Receive <election~~, *ID*~~> from process *p*

- ~~If ID < my ID~~ (always true)
  - Send <disagree> to *p*
  - Start election (if not already running)

Example: Bully Election



Analysis of The Bully Algorithm

Best case scenario: The process with the second highest id notices the failure of the coordinator and elects itself.

- Bandwidth overhead:
  - N-2 coordinator messages are sent
- Turnaround time
  - A single message transmission

Analysis of The Bully Algorithm

Worst case scenario: When the process with the lowest id in the system detects the failure.

Bandwidth overhead

- N-1 processes altogether begin elections, each sending messages to processes with higher ids.
- The message overhead is $O(N^2)$.

Turnaround time

All messages arrive within T units of time (synchronous)

Turnaround time:

- Election message from lowest process (T)
- Timeout at 2nd highest process (X)
- Coordinator message from 2nd highest process (T)

How long should the timeout be?

- X = 2T
- Total turnaround time:

How long should election restart timeout be?

- X + T = 3T

Summary

Coordination in distributed systems requires a leader process

- Need to (re-) elect leader process

Need a way to break symmetry

Three Algorithms

- Ring algorithm
- Modified Ring algorithm
- Bully Algorithm

Readings:

- For today's lecture: Section 15.3

Comparison of Algorithms

Ricart-Agrawala, Bully, ISIS all follow the same pattern:

- multicast to share own state
- wait for responses
- break ties

	ISIS	Ricart-Agrawala	Bully
State multicast	New message, final priority	Request for CS	Election, Coordinator
Response	Proposed priority	CS available	Disagree
Break ties	Priority, pid	Lamport timestamp, pid	pid