Leader Election

CMPS 4760/6760: Distributed Systems

Acknowledgement: slides adapted from Indranil Gupta’s lecture notes:
https://courses.engr.illinois.edu/cs425/fa2019/index.html

Why Election?

- Ex. 1: Your Bank account details are replicated at a few servers, but one of these servers is responsible for receiving all reads and writes, i.e., it is the leader among the replicas
  - What if there are two leaders per customer?
  - What if servers disagree about who the leader is?
  - What if the leader crashes?

Each of the above scenarios leads to Inconsistency

Why Election?

- Ex. 2: Electing a Coordinator
  - E.g., centralized mutual exclusion
- Ex. 3: Breaking symmetry
  - E.g., remove one of the nodes in the cycle to remove the deadlock

Leader Election Problem

- In a group of processes, elect a *Leader* to undertake special tasks
  - And *let everyone know* in the group about this Leader
- What happens when a leader fails (crashes)
  - Some process detects this (using a Failure Detector, see 15.1)
  - Then what?
- Focus of this lecture: Election algorithm. Its goal:

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

System Model

- *N* processes
- Each process has a unique identifier
  - the ‘identifier’ may be any useful value, as long as they are unique and totally ordered, e.g., IP address, <1/load, i>
- Messages are *eventually* delivered
  - the network may be partitioned in any particular interval of time, but
  - a reliable communication protocol masks channel failures
- Failures may occur during the election protocol
  - Processes fail only by crashing

Problem Specification

- Any process can call for an election
  - E.g., when it detects the leader has failed
  - a process can call for at most one election at a time
- Multiple processes can call for elections concurrently
  - All of them together must yield only a single leader
  - The result of an election should not depend on which process calls for it
- Without loss of generality, we require that the elected process be chosen as the one with the largest identifier

Correctness

- At any time, a process is either engaged in some run of the election algorithm (a participant) or not currently engaged in any election (a non-participant)
- Each process has a local variable $elected$ that defines the leader. When a process first becomes a participant, $elected = \perp$ (null)
- Safety: for all non-faulty and participant processes, $elected = (P$: a particular non-faulty process with the largest id) or $\perp$
- Liveness: all non-faulty processes eventually participate and $elected \neq \perp$

Performance

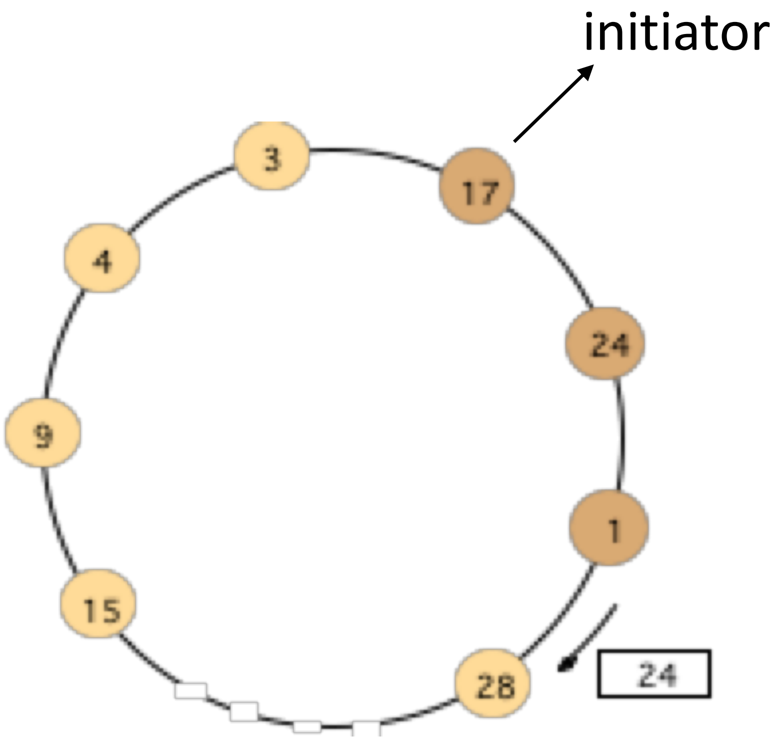
- Network bandwidth utilization: total number messages sent
- Turnaround time: number of serialized message transmission times between the initiation and termination of a single run

Lead Election vs. Mutual Exclusion

- Similarity: whichever process enters the critical section becomes the leader
- Differences
  - Starvation/fairness is irrelevant in leader election
  - Exit from CS is unnecessary for leader election.
  - Leader needs to inform every active process about its identity

Ring Election (Chang-Roberts Algorithm)

- Assumptions
  - Processes arranged in a logical ring: i-th process P_i has a communication channel to $P_{(i+1) \bmod N}$
  - All messages are sent clockwise around the ring
  - No failures (during election)
  - Asynchronous systems
- Main idea: the process with the maximum id gets elected as the leader



Chang-Roberts Algorithm

$\boldsymbol{P_i}$::

var

$myid$: integer;

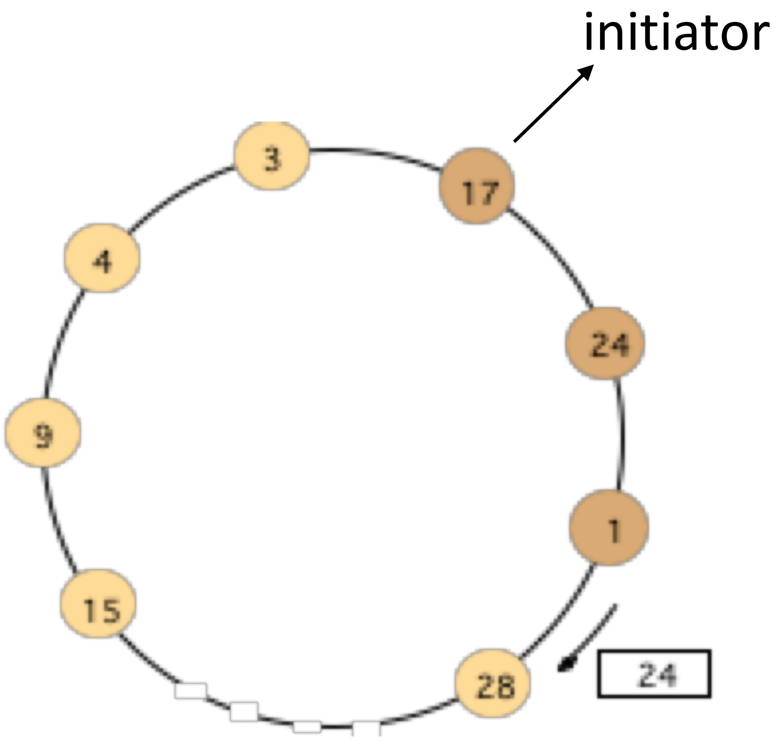
$participant$: boolean initially $false$;

$elected$: integer initially $null$;

To initiate election:

send (Election, $myid$) to P_{i+1};

$participant = true$;



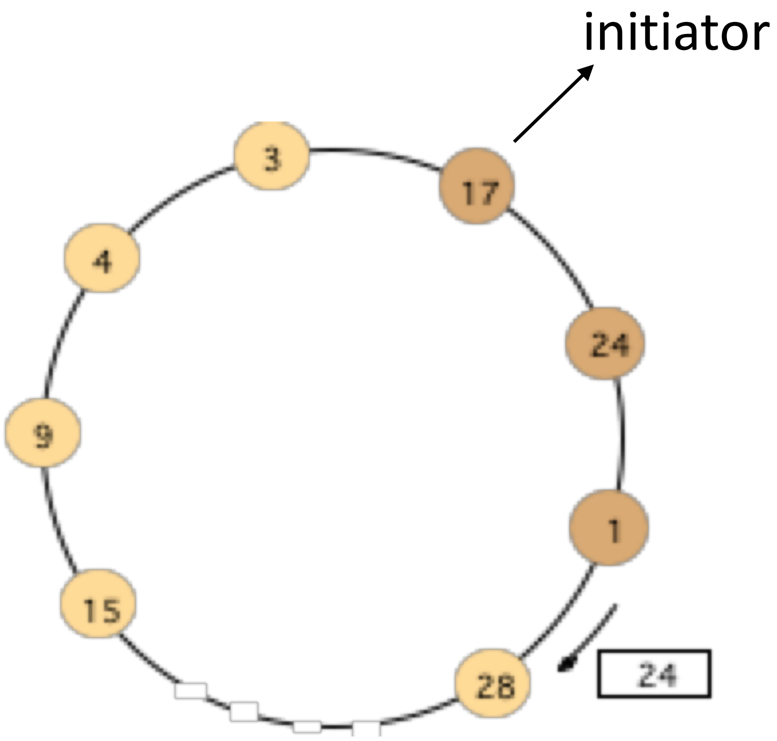
Chang-Roberts Algorithm

Upon receiving a messae $(\text{Election}, j)$:

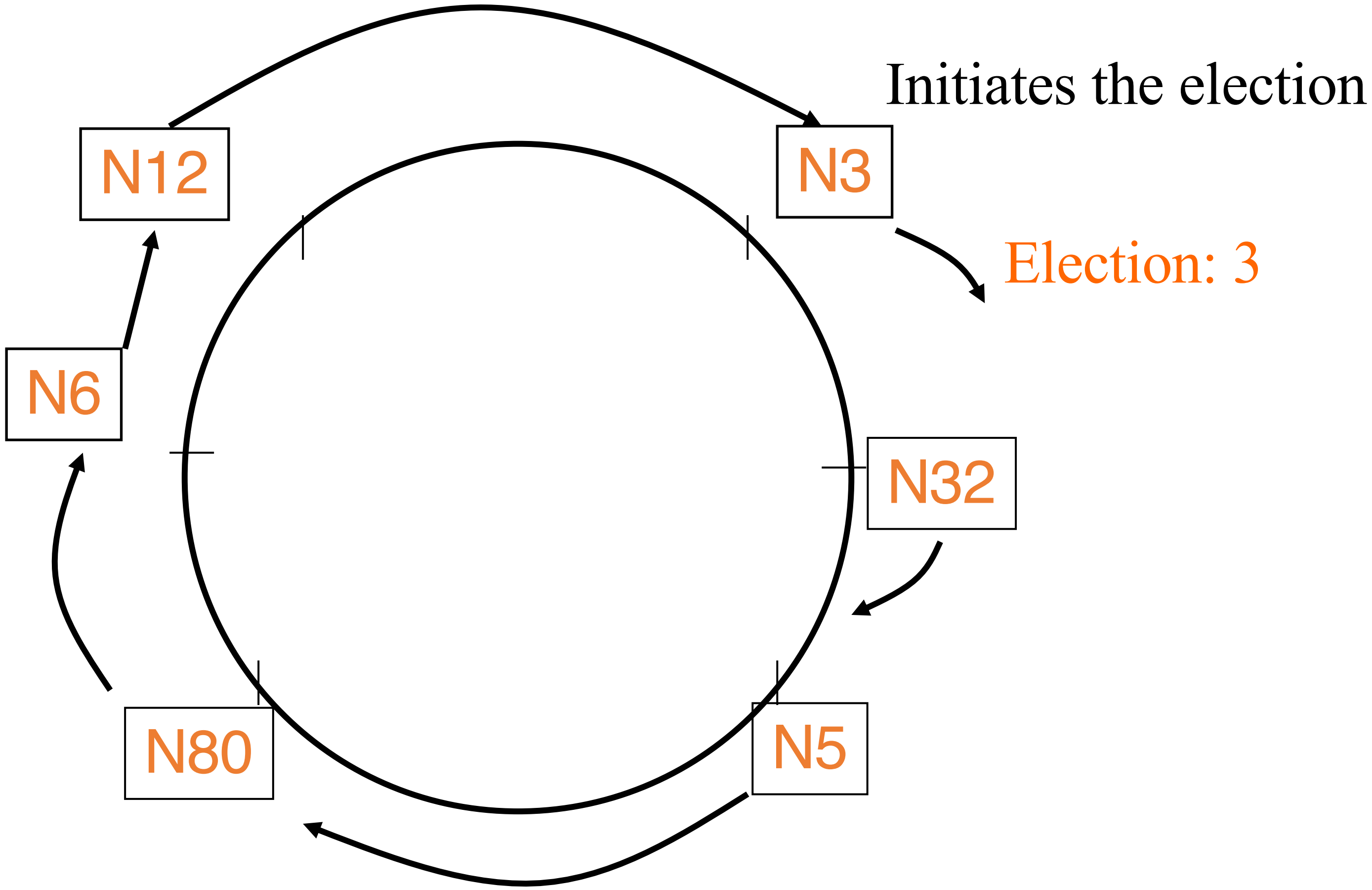
if $(j > myid)$
  send $(\text{Election}, j)$ to P_{i+1};
  $participant = true$;
else if $((j < myid) \wedge \neg participant)$
  send $(\text{Election}, myid)$ to P_{i+1};
  $participant = true$;
else if $(j == myid)$
  send $(\text{Elected}, myid)$ to P_{i+1};

Upon receiving a messae $(\text{Elected}, j)$:

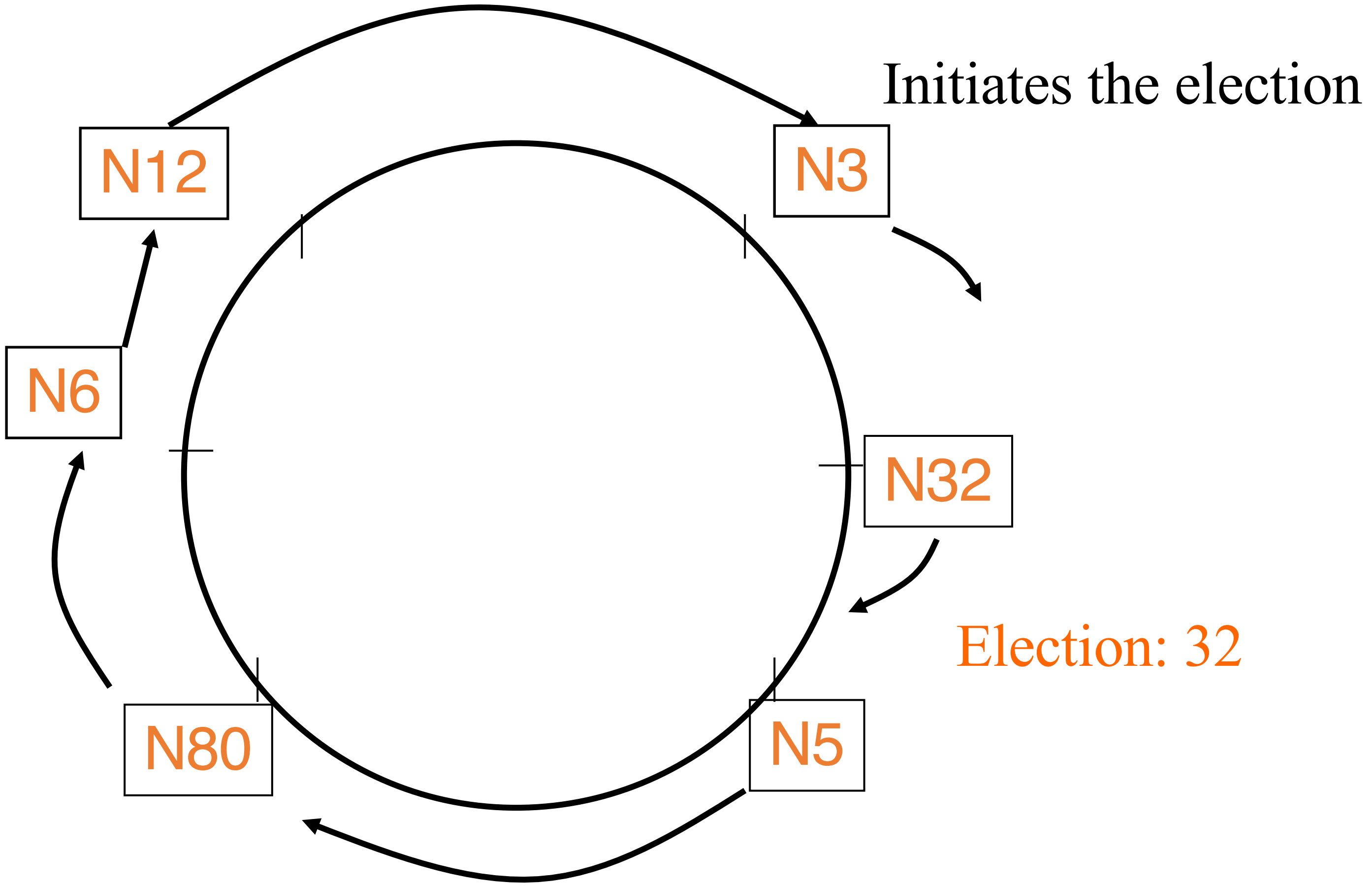
$elected = j$;
if $(j \neq myid)$ send $(\text{Elected}, j)$ to P_{i+1};



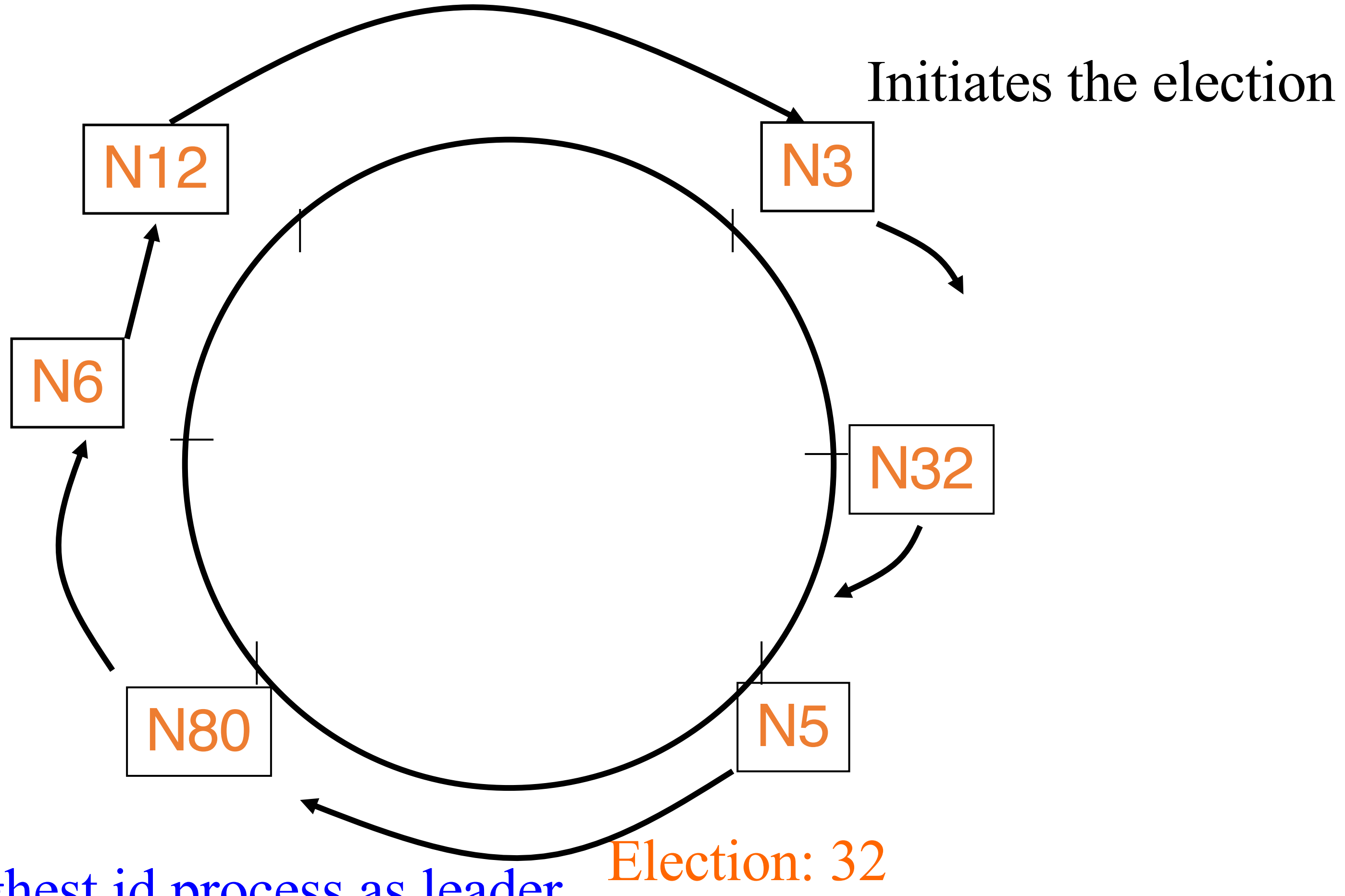
Ring Election: Example



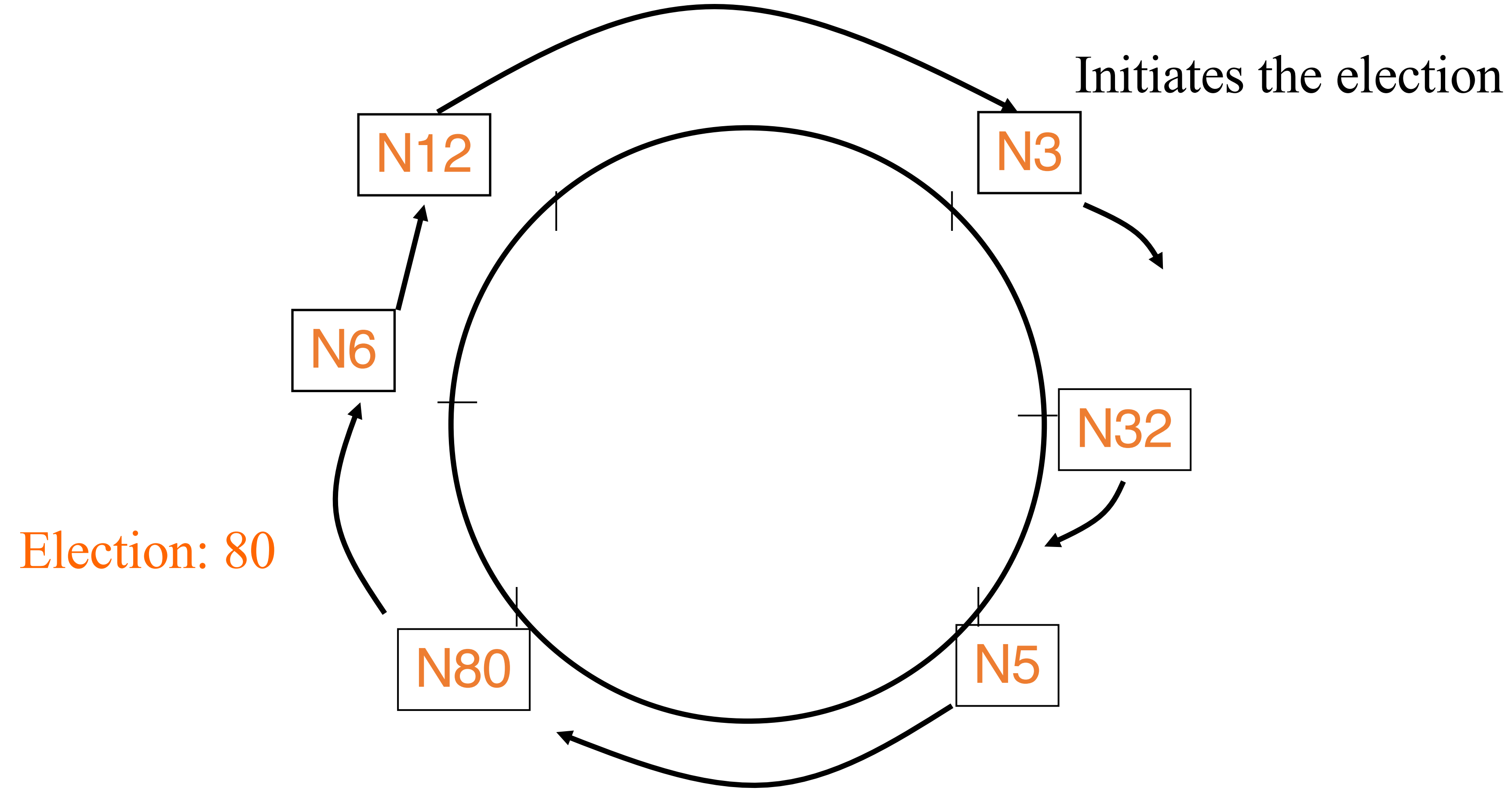
Goal: Elect highest id process as leader



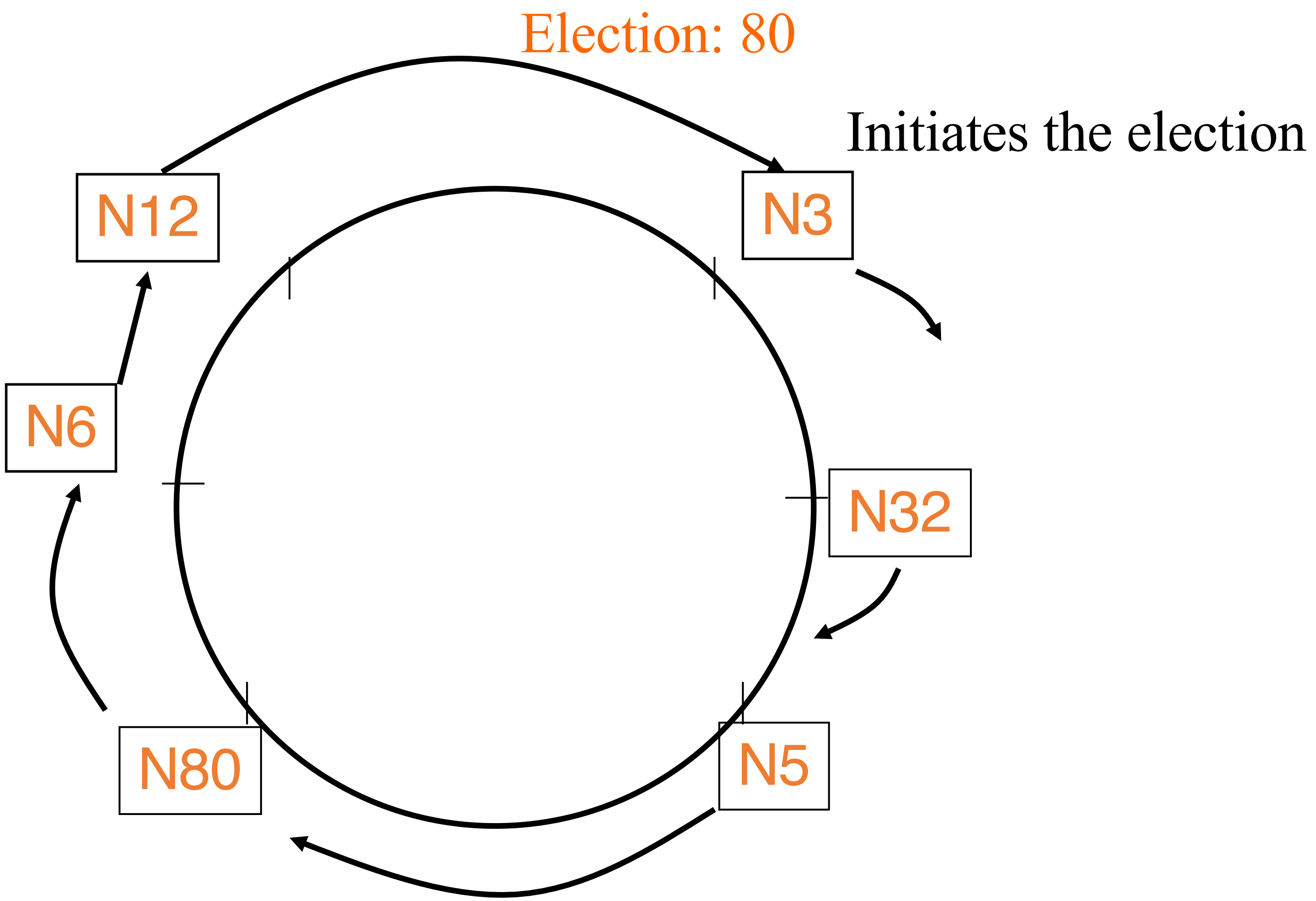
Goal: Elect highest id process as leader



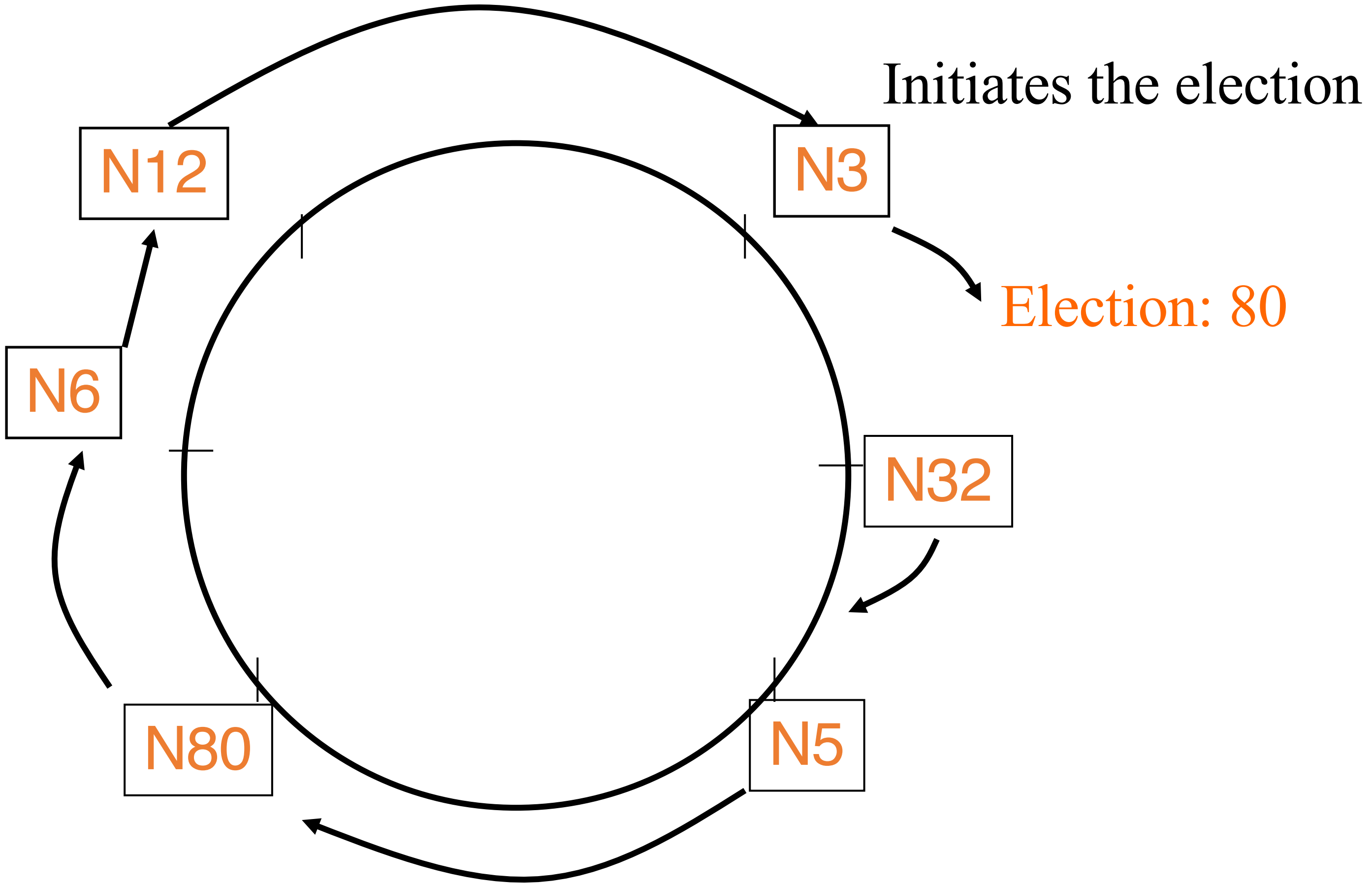
Goal: Elect highest id process as leader



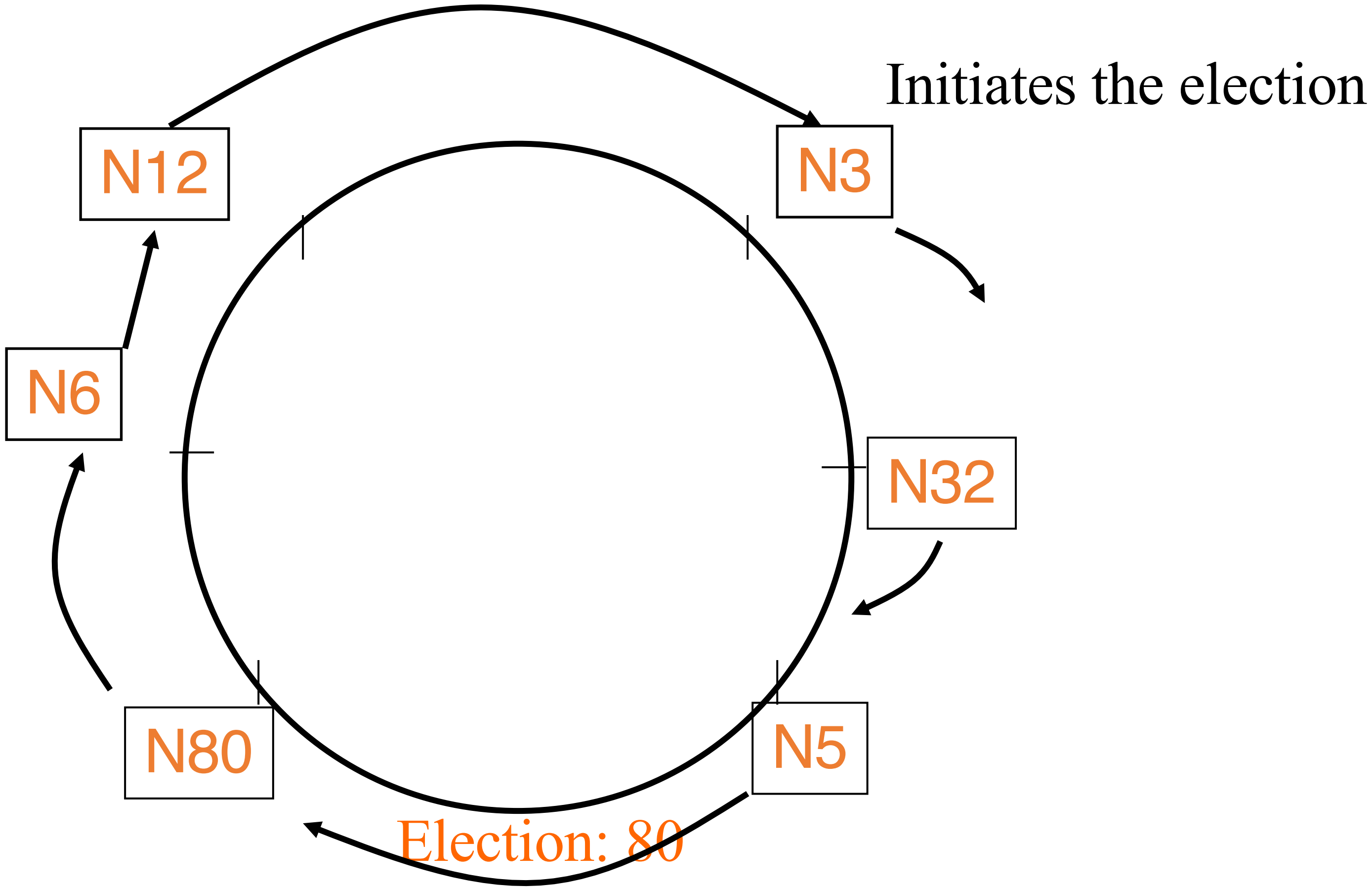
Goal: Elect highest id process as leader



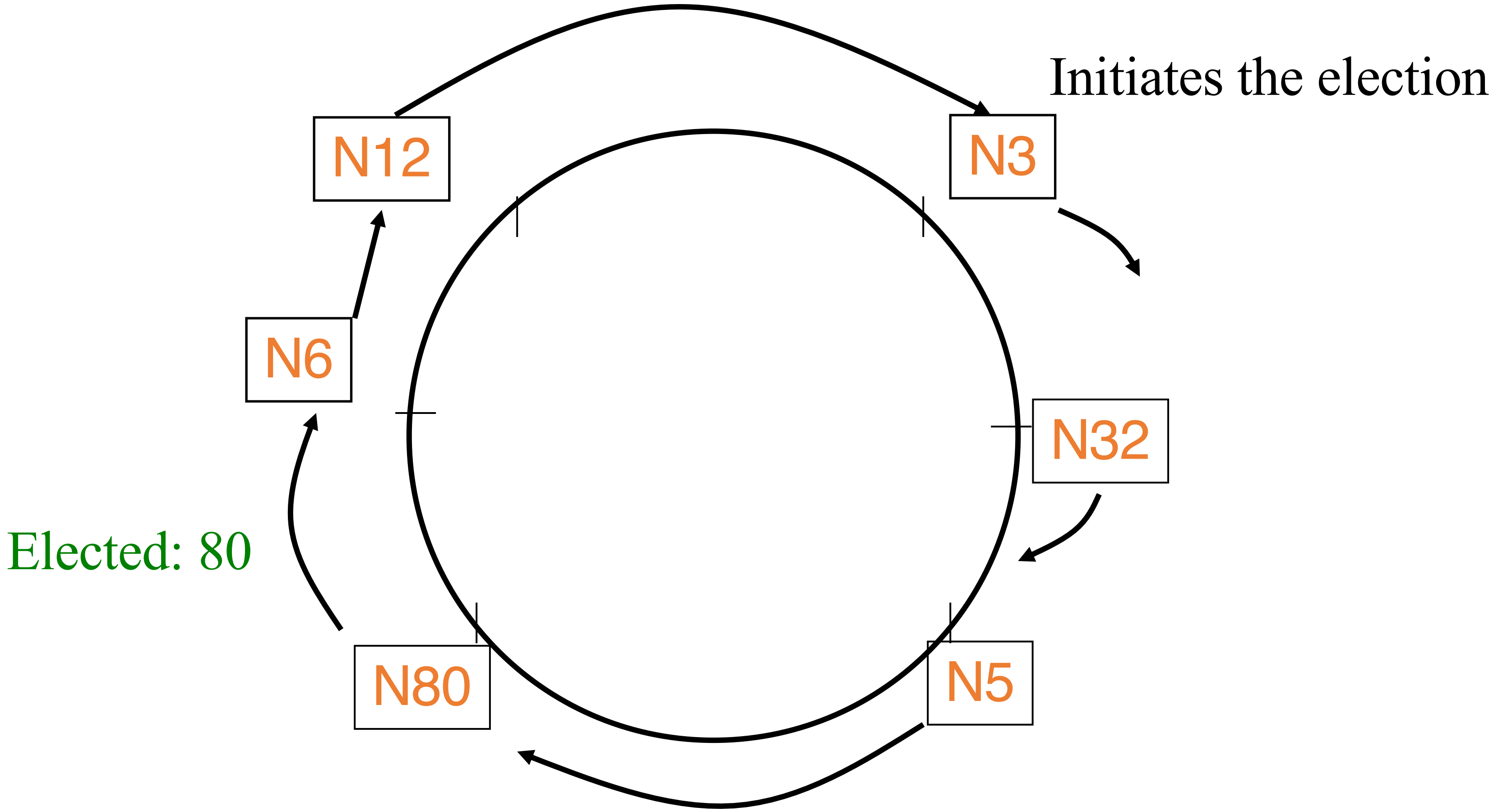
Goal: Elect highest id process as leader



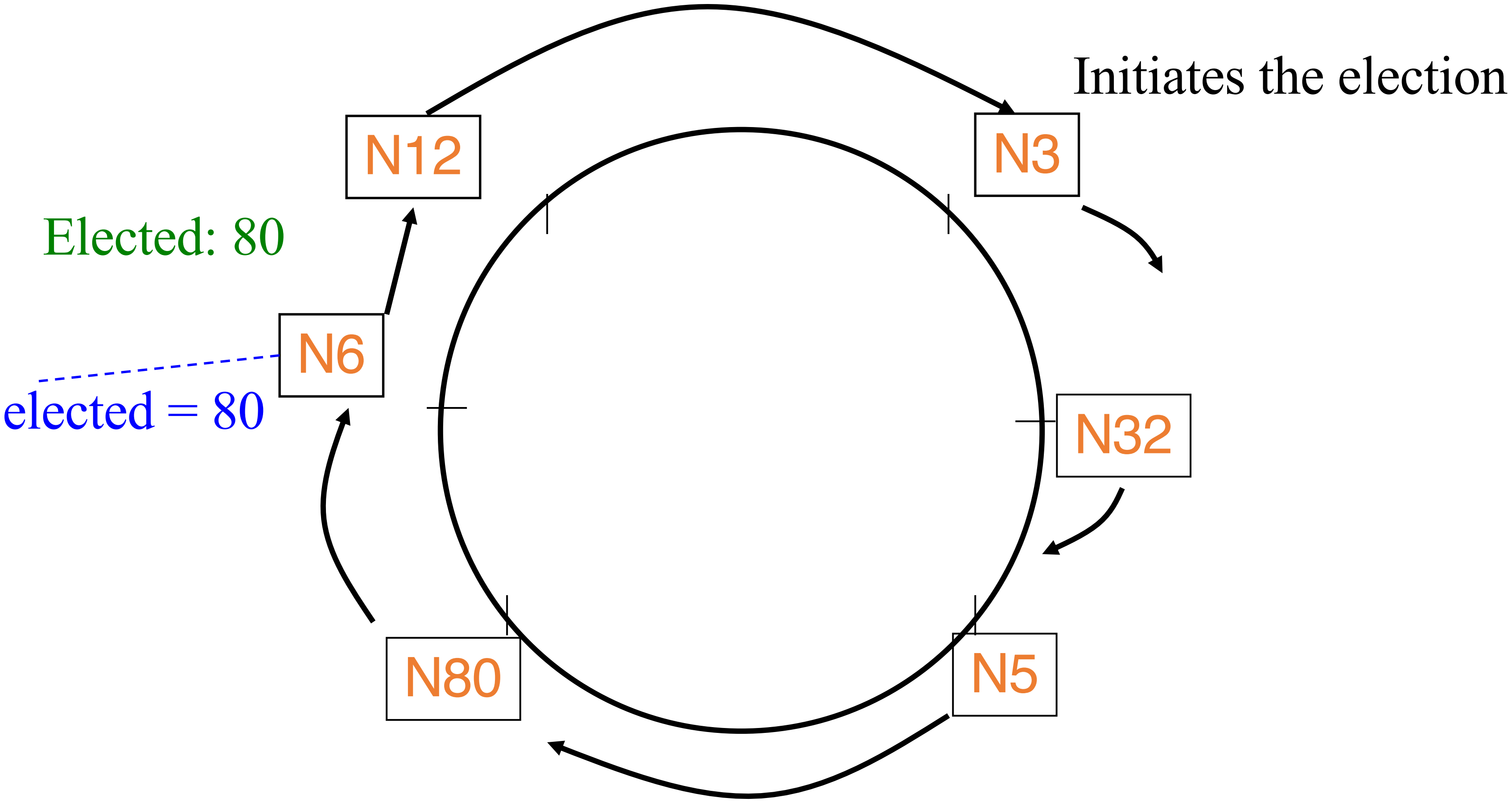
Goal: Elect highest id process as leader



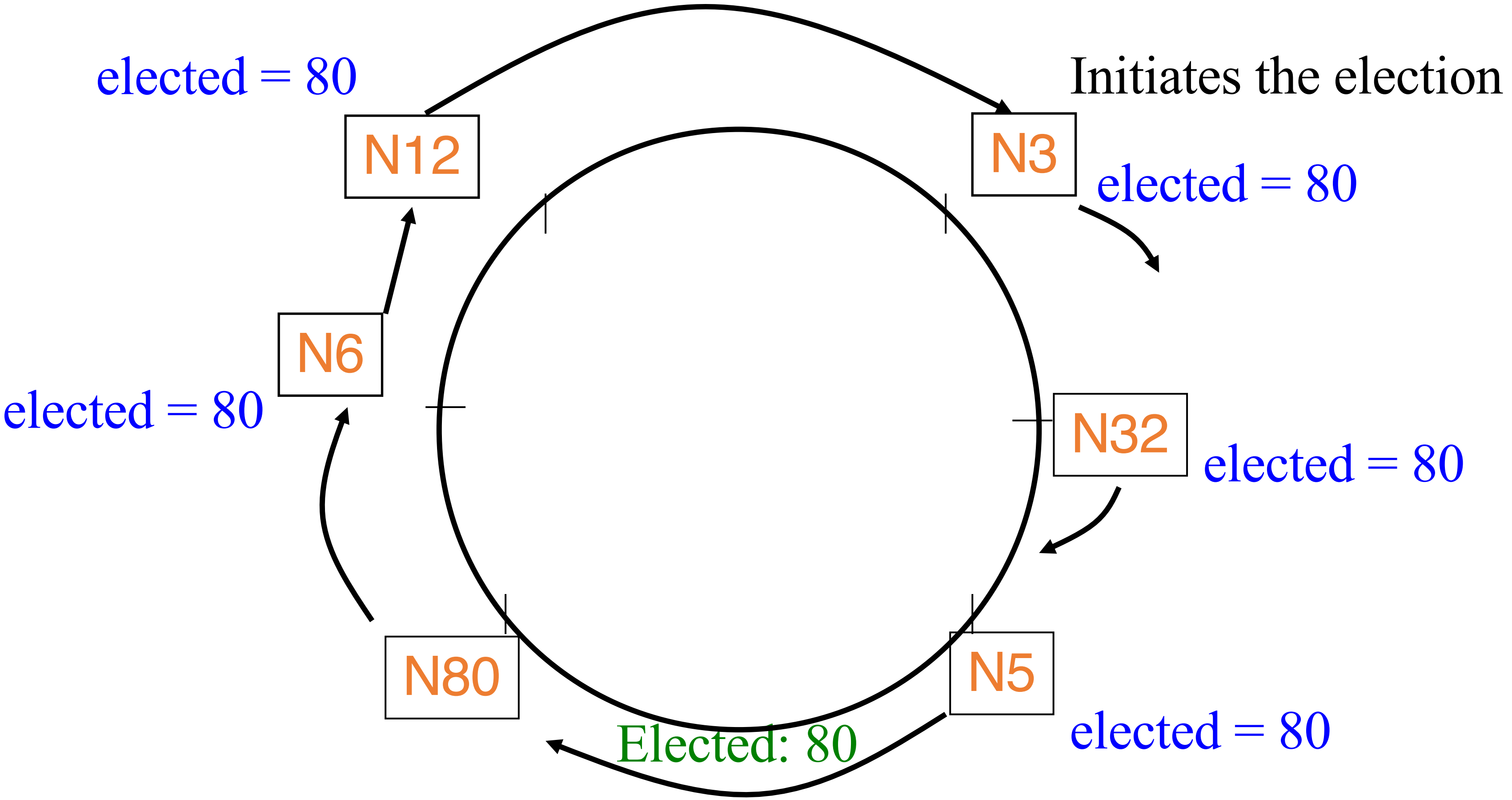
Goal: Elect highest id process as leader



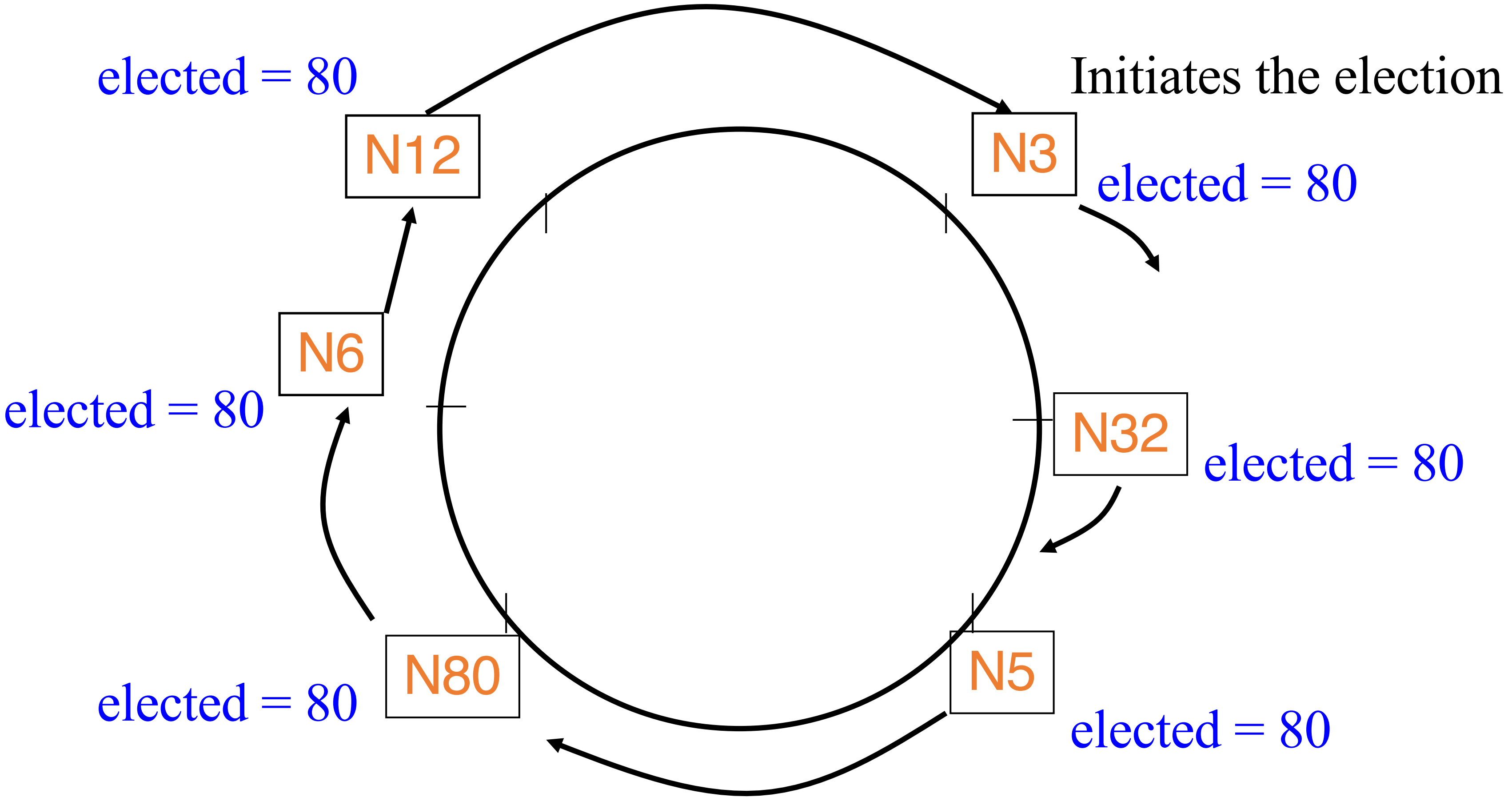
Goal: Elect highest id process as leader



Goal: Elect highest id process as leader



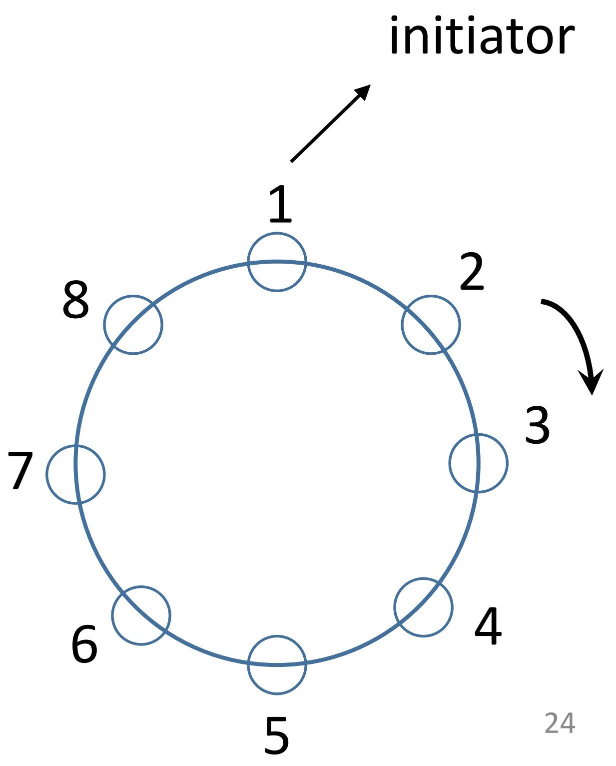
Goal: Elect highest id process as leader



Goal: Elect highest id process as leader

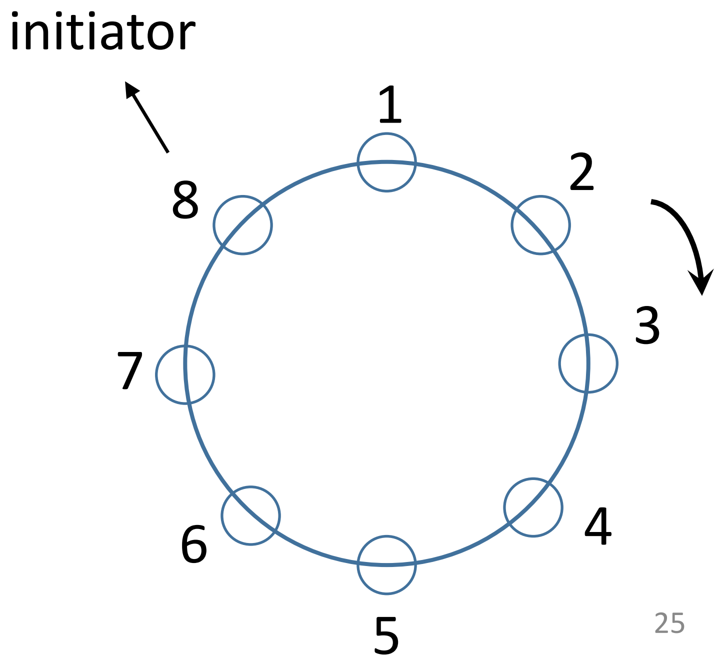
Analysis

- Safety and liveness satisfied
- Performance (single initiator)
  - Worst case
    - The anti-clockwise neighbor of the initiator has the highest id
    - $N-1$ *Election* messages to reach this neighbor
    - Another N *Election* messages before it announces its election
    - N *Elected* message
    - Message complexity: $3N-1$ messages
    - Turnaround time: $3N-1$ message transmission times



Analysis (2)

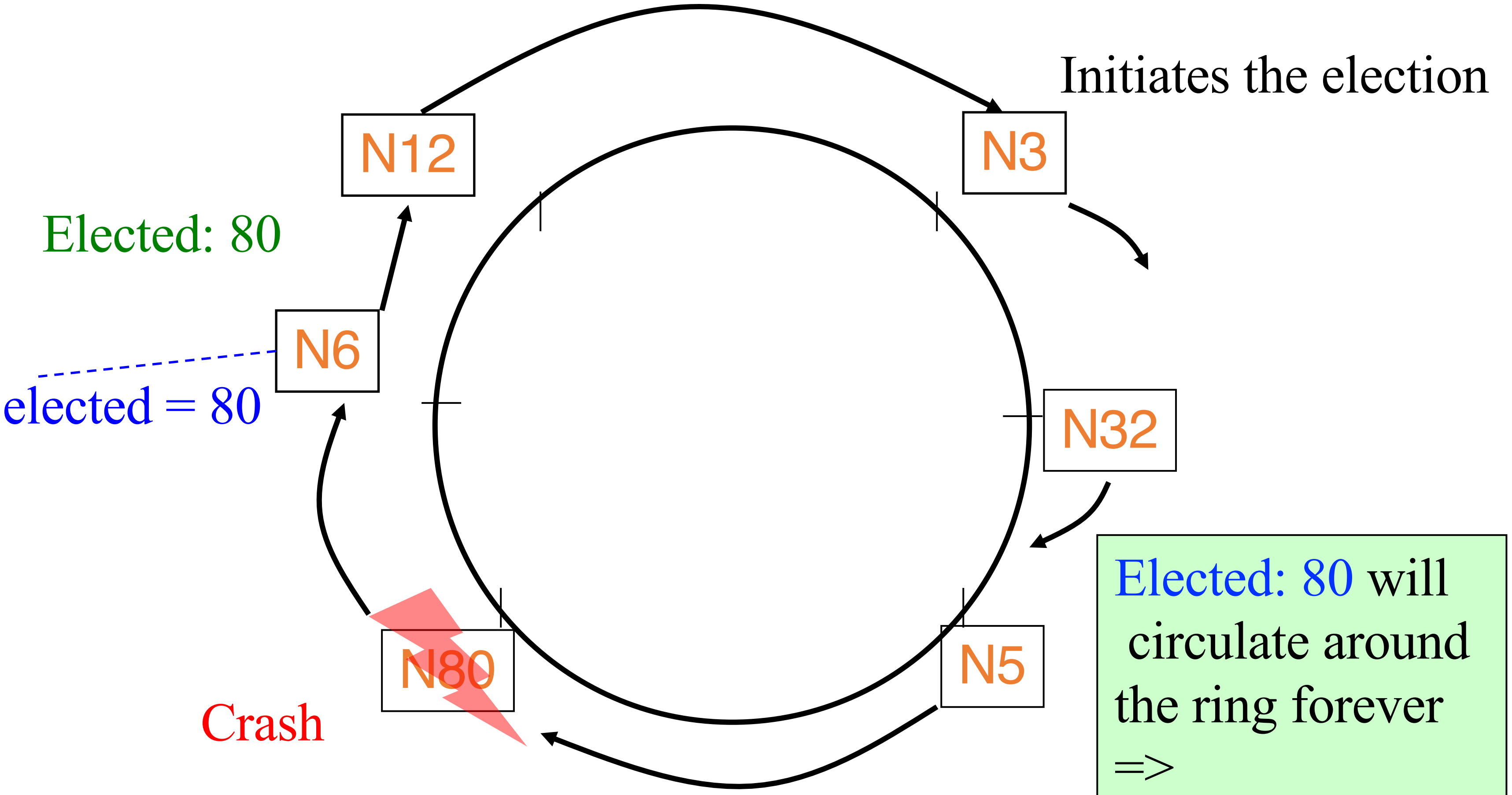
- Safety and liveness satisfied
- Performance (single initiator)
  - Best case
    - Initiator is the would-be leader
    - N *Election* messages
    - N *Elected* message
    - Message complexity: $2N$ messages
    - Turnaround time: $2N$ message transmission times



Multiple Initiators?

- Include initiator's id with all messages
- Each process remembers in cache the initiator of each Election/Elected message it receives
- (All the time) Each process suppresses Election/Elected messages of any lower-id initiators
- Updates cache if receives higher-id initiator's Election/Elected message
- Result is that only the highest-id initiator's election run completes

Effect of Failures



Elected: 80 will circulate around the ring forever
=>
Liveness violated

Fixing for failures

- One option: have predecessor (or successor) of would-be leader N80 detect failure and start a new election run
  - May re-initiate election if
    - Receives an Election message but times out waiting for an Elected message
    - Or after receiving the Elected:80 message
  - But what if predecessor also fails?
  - And its predecessor also fails? (and so on)

Fixing for failures (2)

- Second option: any process, after receiving Election:80 message, can detect failure of N80 via its own local failure detector
  - If so, start a new run of leader election
- But failure detectors may not be both complete and accurate
  - Completeness = each failure is detected
  - Accuracy = there is no mistaken detection
  - Incompleteness in FD => N80's failure might be missed
  - Inaccuracy in FD => N80 mistakenly detected as failed => new election runs initiated forever

Why is Election so Hard?

- Because it is related to the consensus problem!
- If we could solve election, then we could solve consensus!
  - Elect a process, use its id’s last bit as the consensus decision
- But since consensus is impossible in asynchronous systems with failures, so is election!

- Consensus-like protocols used in industry for leader election

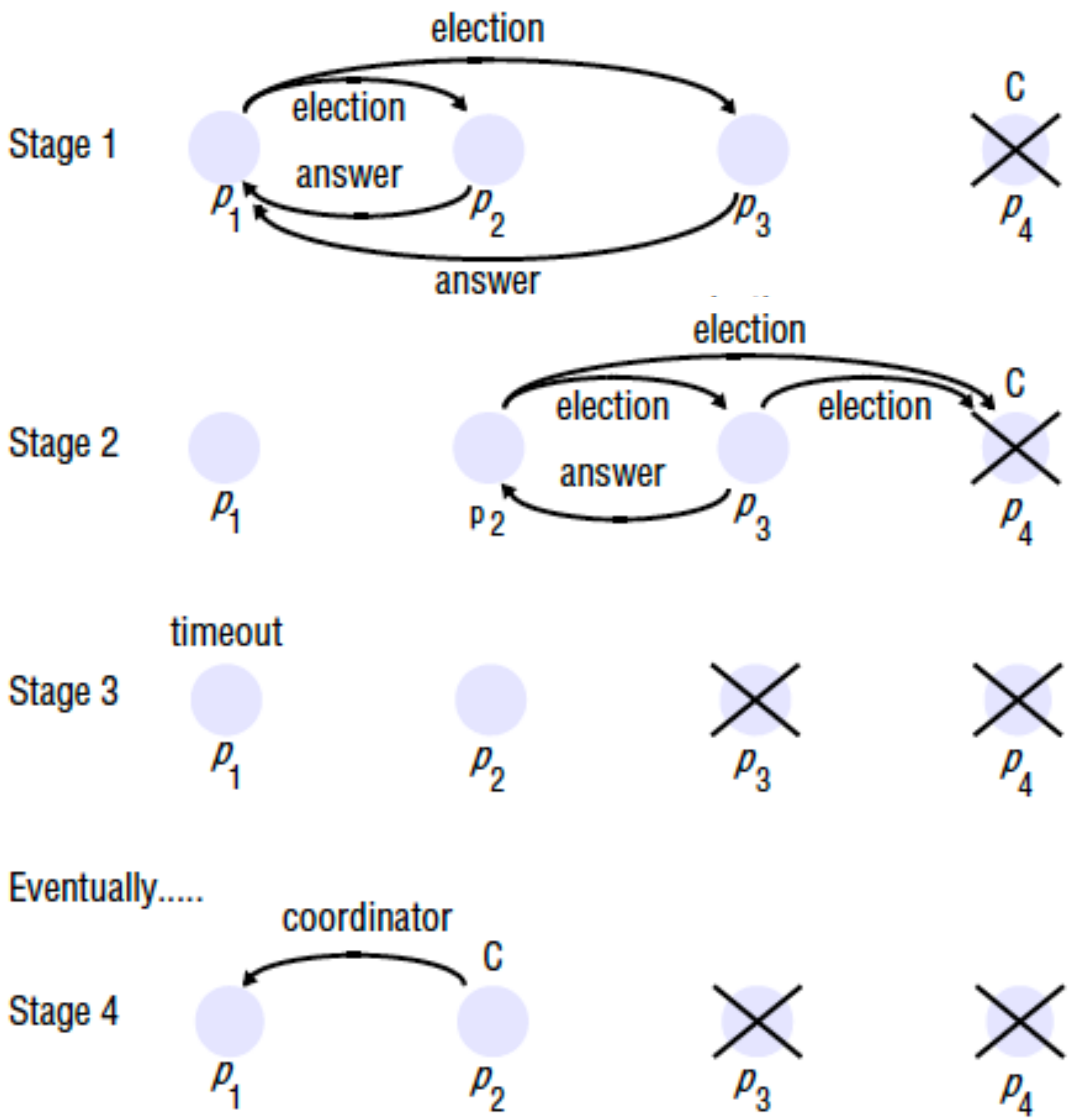
Bully Algorithm

- Assumptions
  - Processes can crash, channels are reliable
  - Synchronized systems: can detect process failures via timeouts
    - Timeout: $T = 2T_{trans} + T_{process}$
  - A completely connected graph
  - Each process knows other processes and their identifiers.

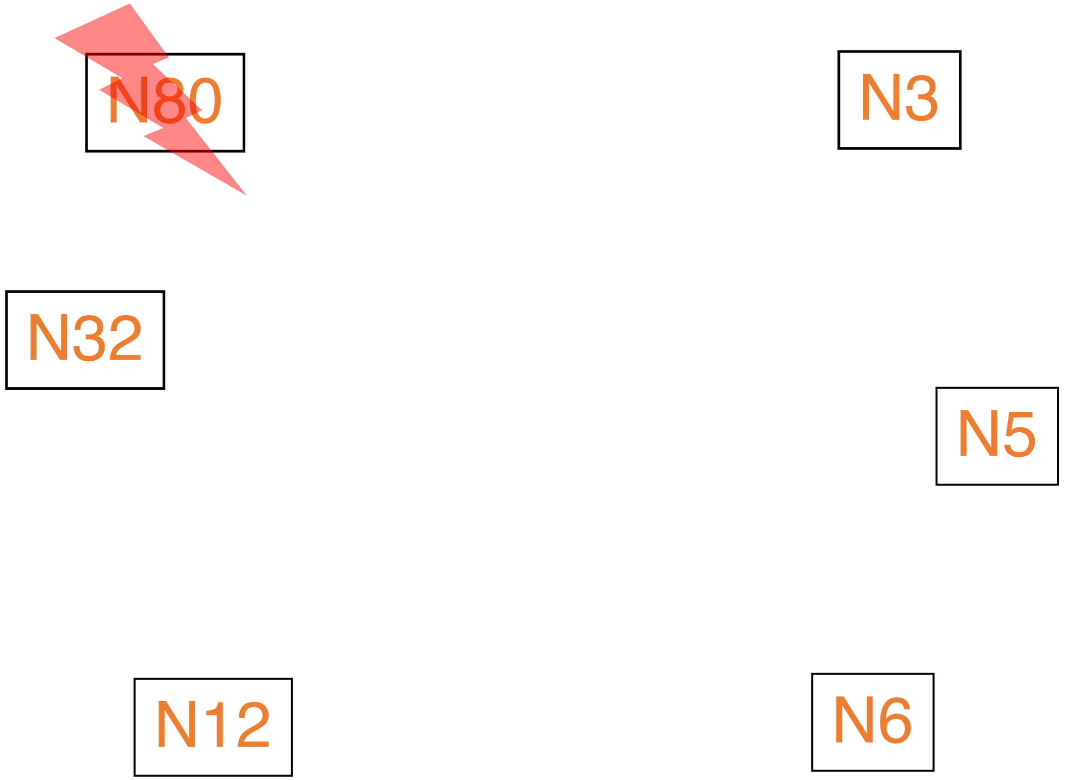
Bully Algorithm

- Any process P can initiate an election (when it notices the leader has failed)
- P sends *election* messages to all processes with higher IDs and awaits *answers*
  - If no *answer* messages arrives within T, P becomes leader and sends *coordinator* messages to all processes with lower IDs
  - If it receives an *answer*, it drops out and waits for a *coordinator* message (if no *coordinator* message with T', restart election)
- If P receives an *election* message
  - Immediately broadcast a *coordinator* message if it is the process with highest ID
  - Otherwise, returns an *answer* message and starts an election (unless it has begun one)
- If P receives a *coordinator* message, it treats sender as the leader

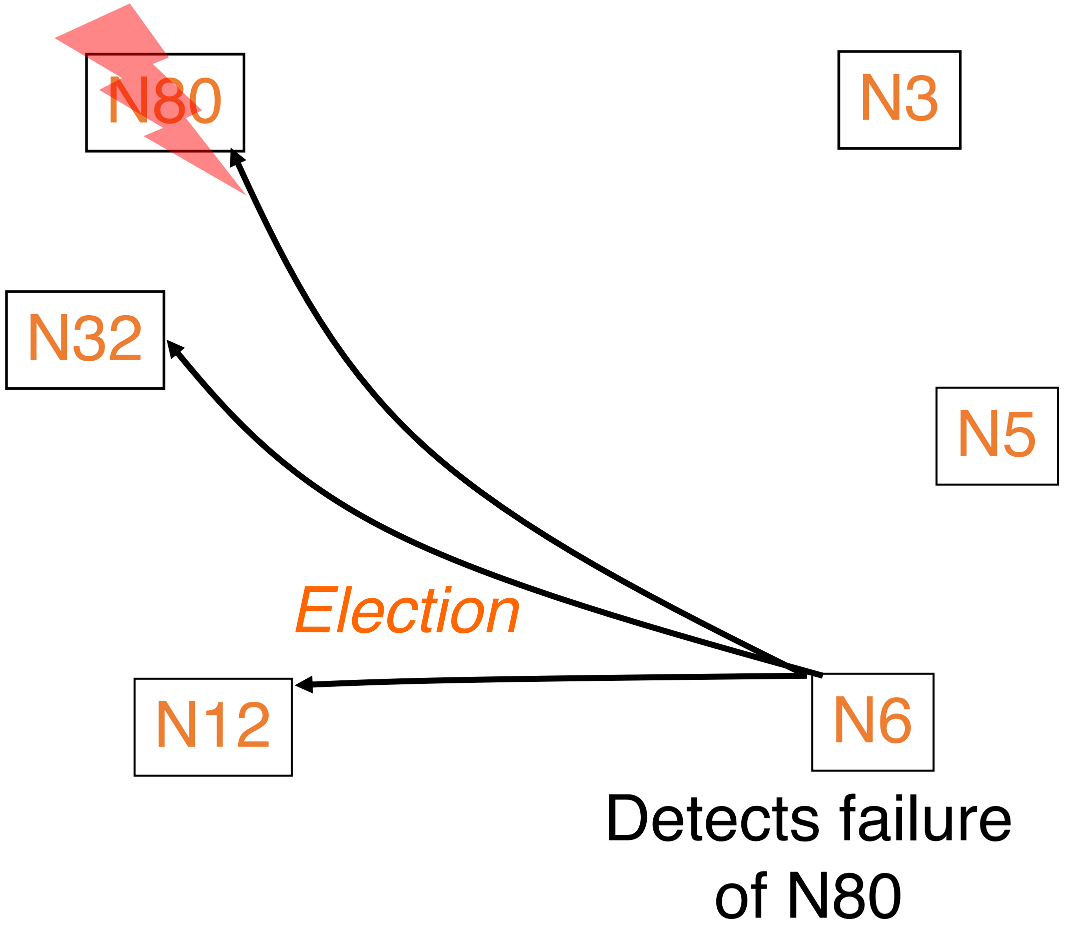
Bully Algorithm

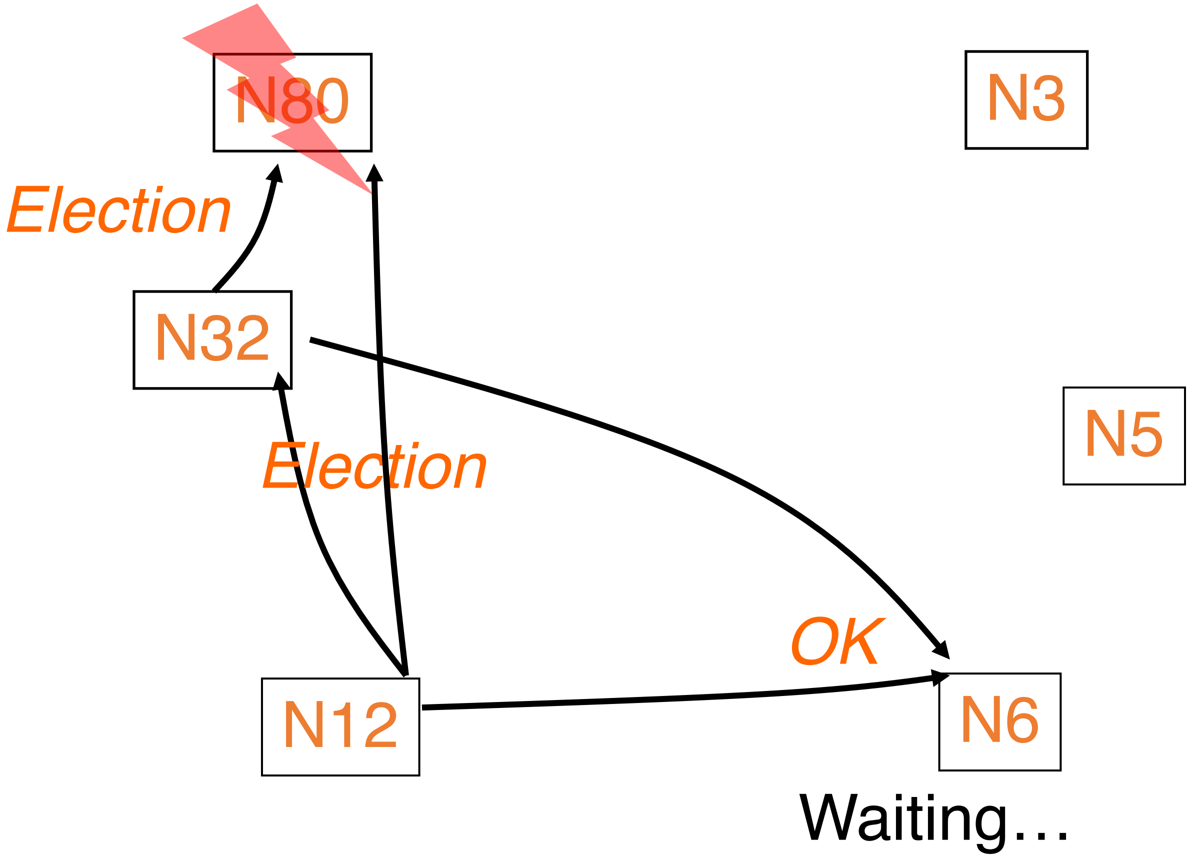


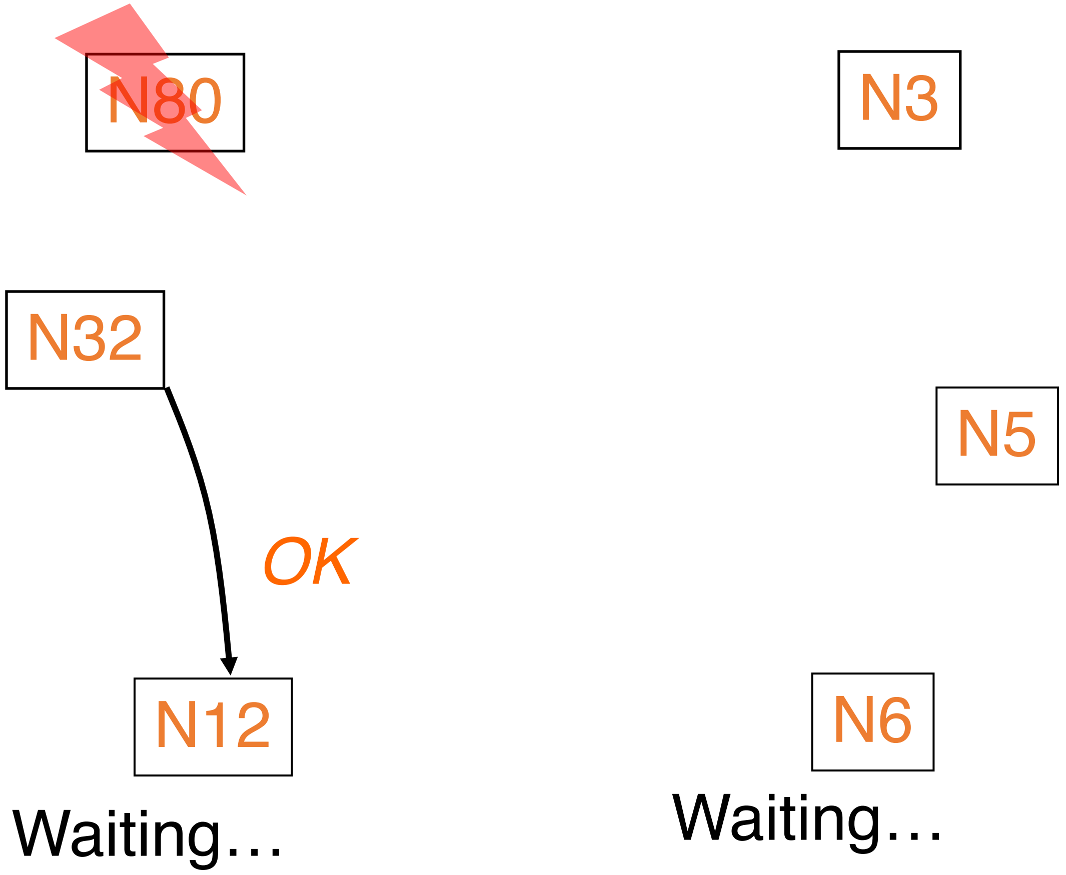
Bully Algorithm: Example

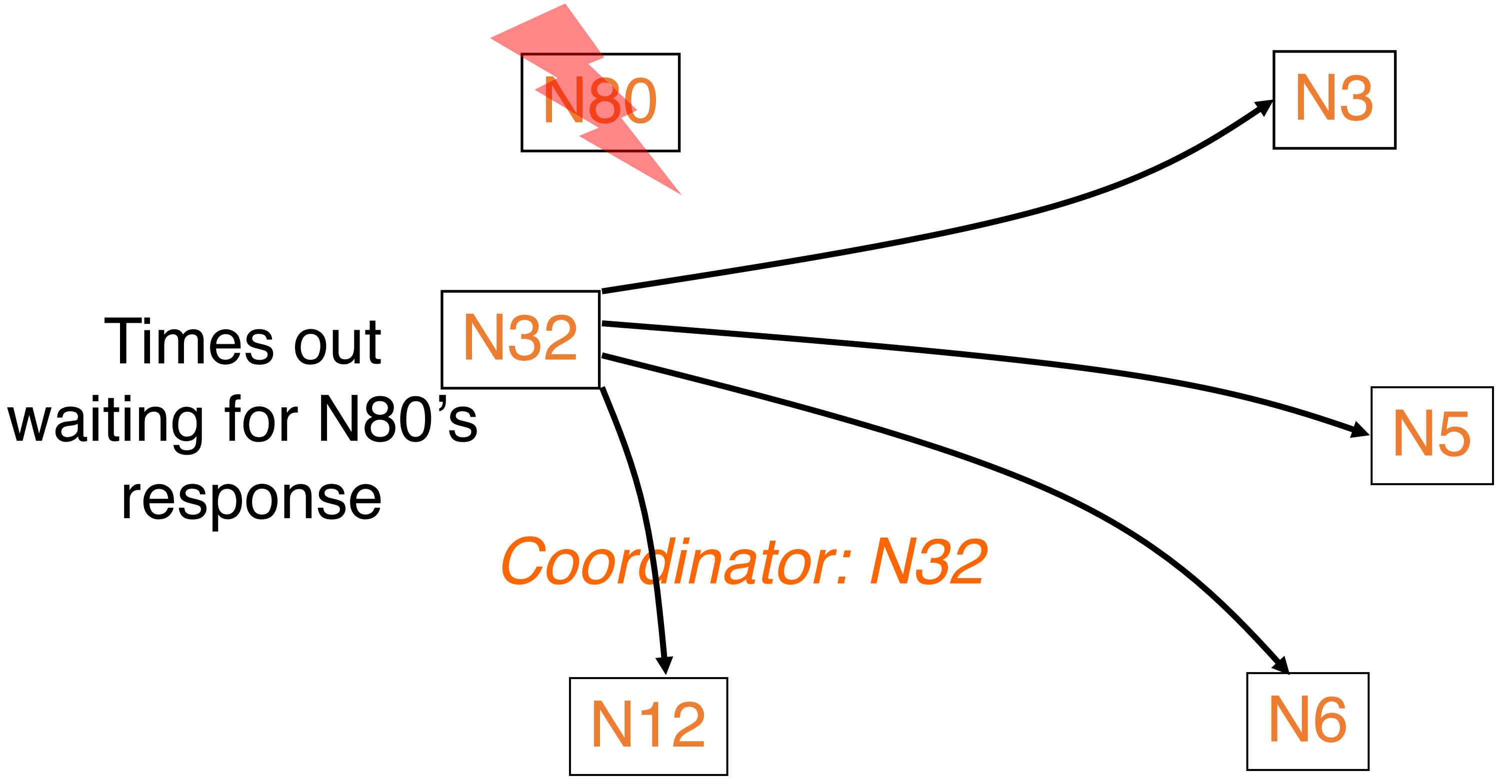


Detects failure
of N80


N80
N3
N32
N5
Election
N12
N6
Detects failure
of N80

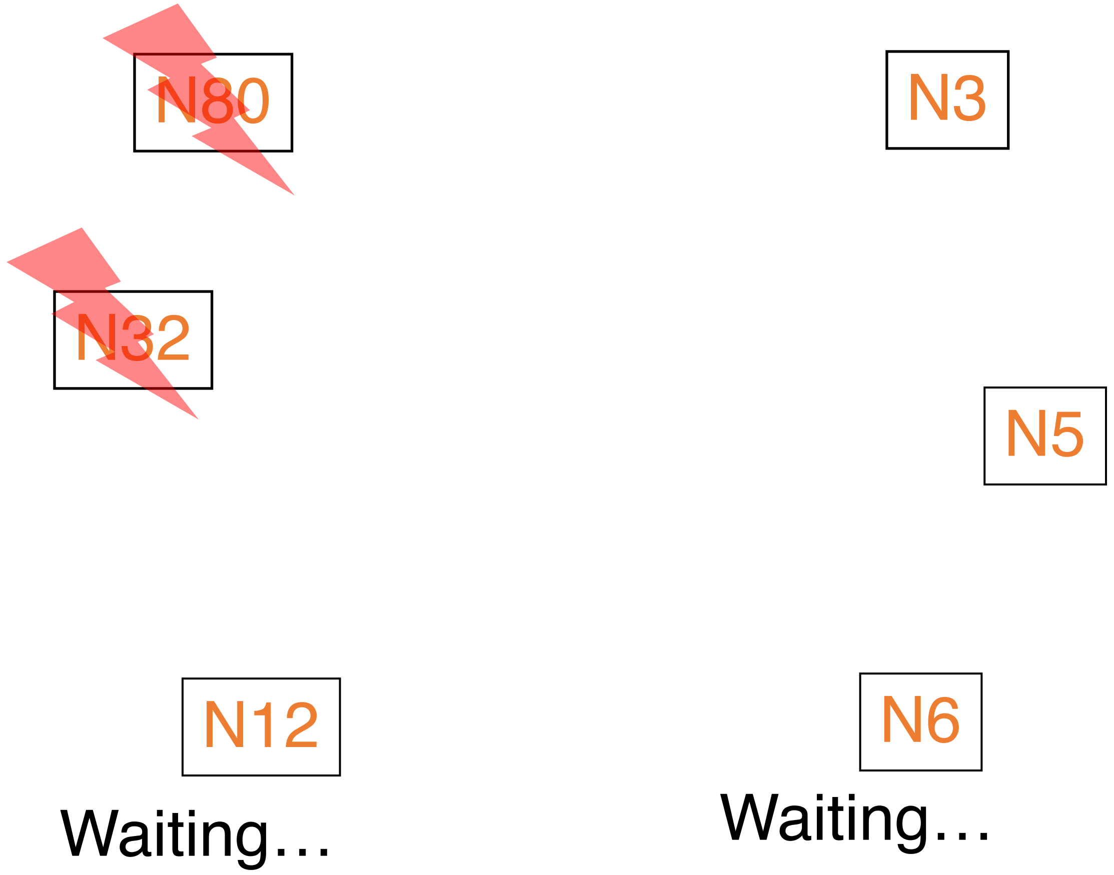

N80
N3
Election
N32
N5
Election
OK
N12
N6
Waiting…

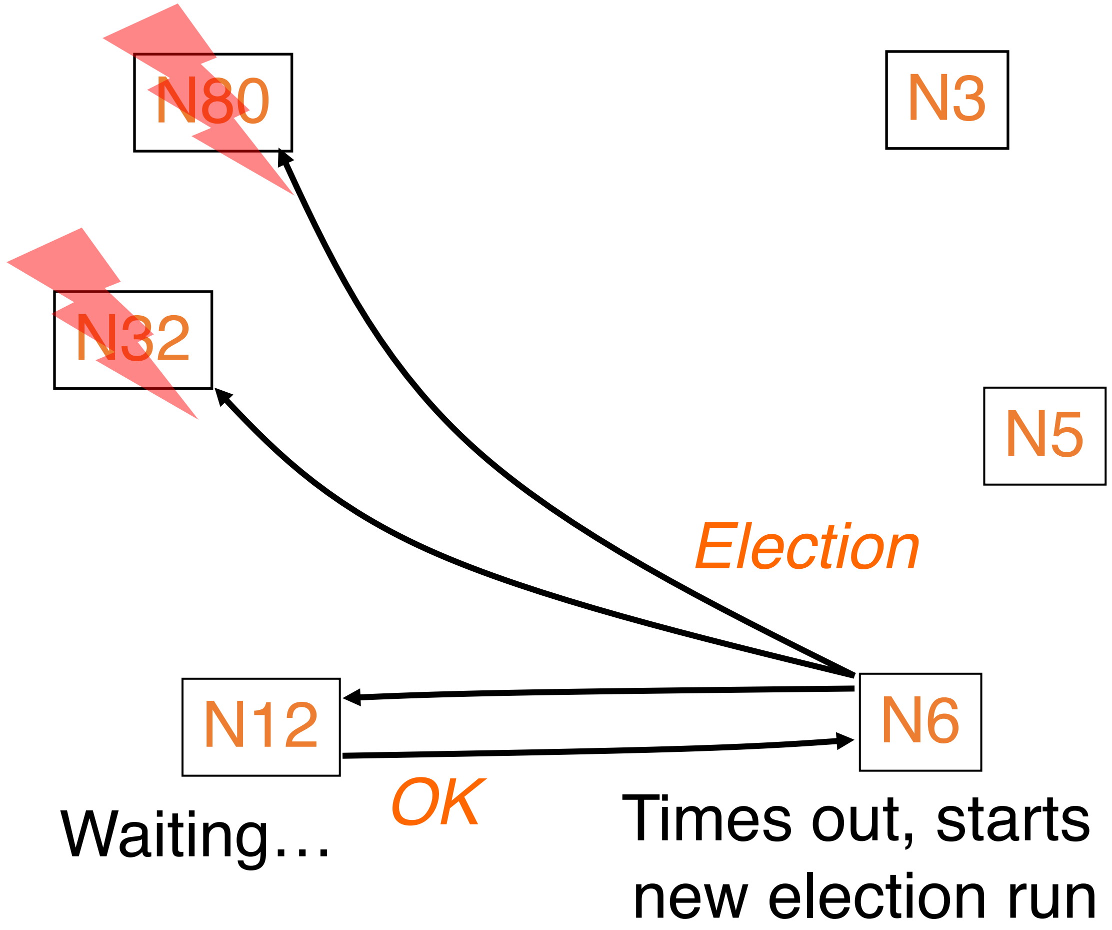

N80
N3
N32
N5
OK
N12
N6
Waiting…
Waiting…

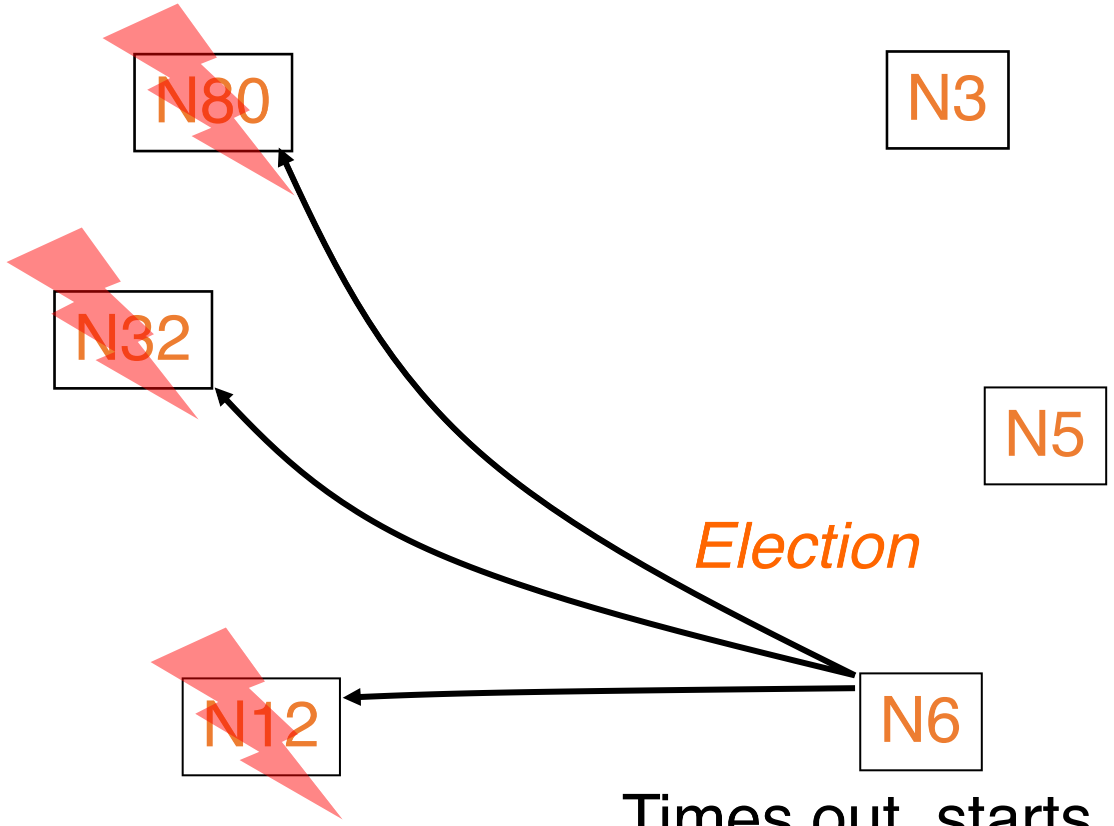


Election is completed

Failures During Election Run




N80
N3
N32
N5
Election
N12
N6
OK
Waiting…
Times out, starts
new election run



Times out, starts
another new election run

Bully Algorithm

- Meets the liveness requirement (in synchronous systems)
- Meets the safety requirement if no process is replaced
- Performance – best case
  - the process with the second highest id notices the failure of the coordinator and elects itself.
  - $N - 2$ *coordinator* messages sent.
  - Turnaround time is one message transmission time.

Bully Algorithm

- Performance – worst case
  - the process with the lowest id detects the failure.
  - $N - 1$ processes altogether begin elections
  - Message complexity is $O(N^2)$
  - Turnaround time: see Homework 2
- 5 message transmission times if there are no failures during the run:
  1. Election from lowest id process in group
  2. Answer to lowest id process from 2^{nd} highest id process
  3. Election from 2nd highest id process to highest id process
  4. Timeout for answers @ 2nd highest id process
  5. Leader from 2^{nd} highest id process

Can use Consensus to solve Election

- One approach
  - Each process proposes a value
  - Everyone in group reaches consensus on some process P_i's value
  - That lucky P_i is the new leader!

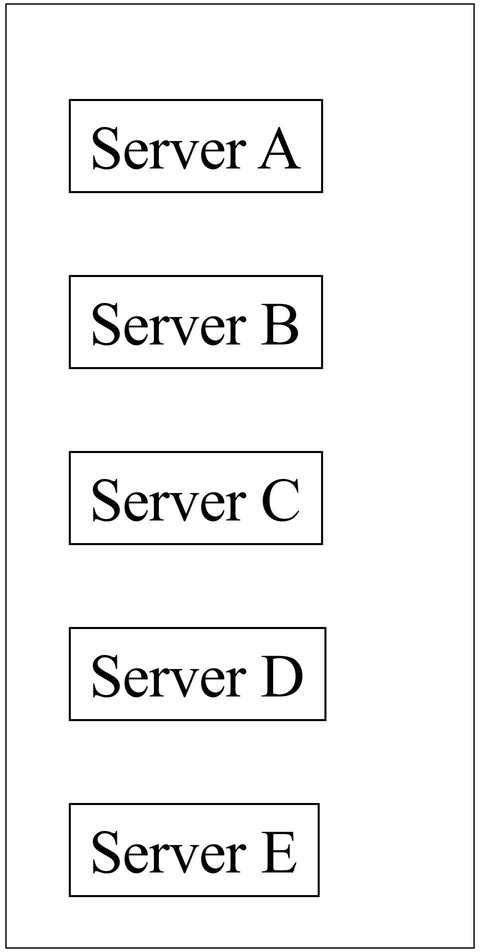
Election in Industry

- Several systems in industry use Paxos-like approaches for election
  - Paxos is a consensus protocol (safe, but eventually live): later in this course
  - Safety: Consensus is not violated
  - Eventual Liveness: If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached. But there is no guarantee.
- Google's Chubby system
- Apache Zookeeper

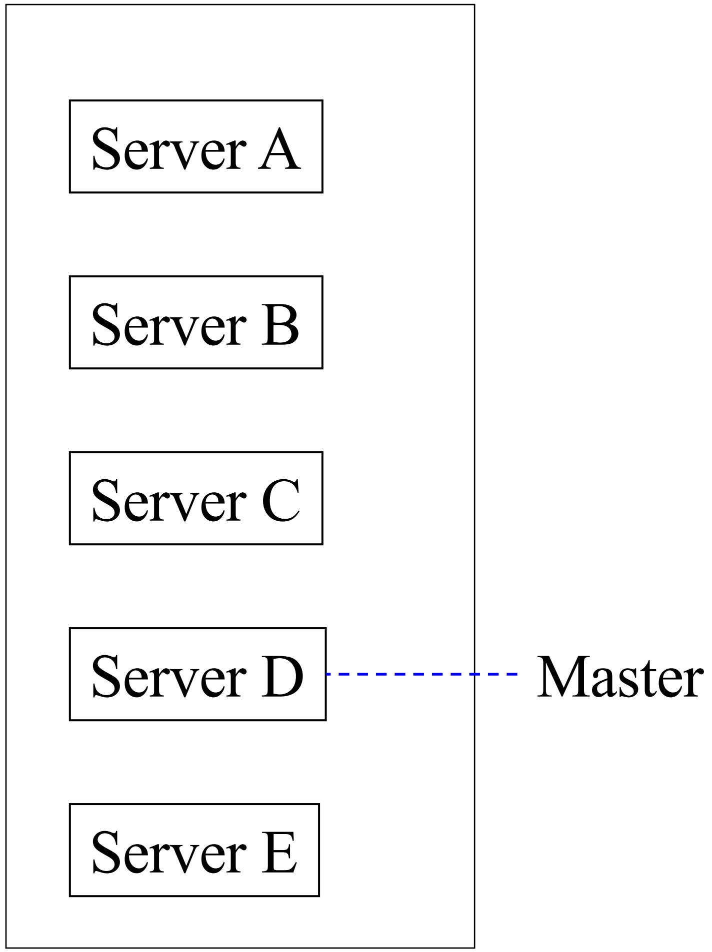
Election in Google Chubby

- A system for locking
- Essential part of Google's stack
  - Many of Google's internal systems rely on Chubby
  - BigTable, Megastore, etc.
- Group of replicas
  - Need to have a master server elected at all times

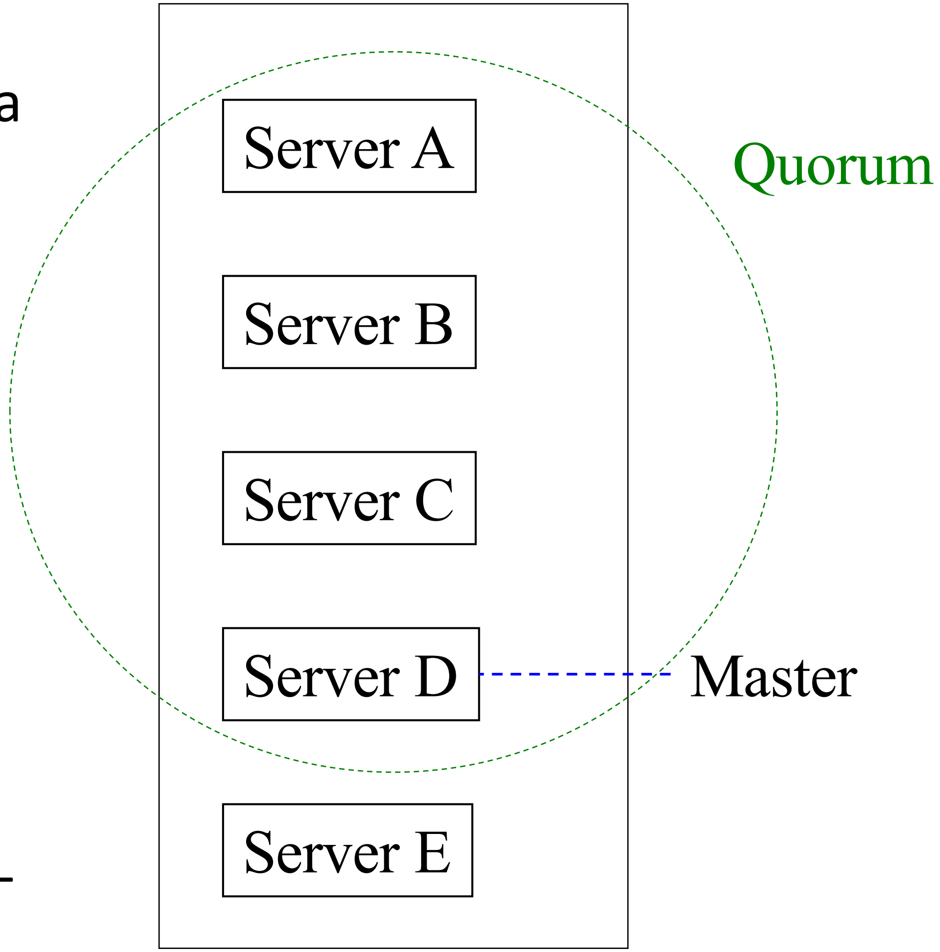
Reference: http://research.google.com/archive/chubby.html



- Group of replicas
  - Need to have a master (i.e., leader)
- Election protocol
  - Potential leader tries to get votes from other servers
  - Each server votes for at most one leader
  - Server with *majority* of votes becomes new leader, informs everyone



- Why safe?
  - Essentially, each potential leader tries to reach a *quorum*
  - Since any two quorums intersect, and each server votes at most once, cannot have two leaders elected simultaneously
- Why live?
  - Only eventually live! Failures may keep happening so that no leader is ever elected
  - In practice: elections take a few seconds. Worst-case noticed by Google: 30s



- After election finishes, other servers promise not to run election again for “a while”
  - “While” = time duration called “Master lease”
  - Set to a few seconds
- Master lease can be renewed by the master as long as it continues to win a majority each time
- Lease technique ensures automatic re-election on master failure