Leader Election in Rings (Classical Distributed Algorithms)

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the leader election problem in message-passing systems for a ring topology, in which a group of processors must choose one among them to be a leader.

- We will present the different algorithms for leader election problem by taking the cases like **anonymous/ non-anonymous rings, uniform/ non-uniform rings and synchronous/ asynchronous rings etc.**

Leader Election (LE) Problem: Introduction

- The leader election problem has several variants.
- LE problem is for each processor to decide that either it is **the leader or non-leader**, subject to the constraint that exactly one processor decides to be the leader.
- LE problem represents a general class of ***symmetry-breaking*** problems.
- For example, when a deadlock is created, because of processors waiting in a cycle for each other, the deadlock can be broken by electing one of the processor as a leader and removing it from the cycle.

Leader Election: Definition

- Each processor has a set of **elected (won)** and **not-elected (lost)** states.
- Once an elected state is entered, processor is always in an elected state (and similarly for not-elected): i.e., irreversible decision
- In every admissible execution:
  - every processor eventually enters either an elected or a not-elected state
  - exactly one processor (the **leader**) enters an elected state

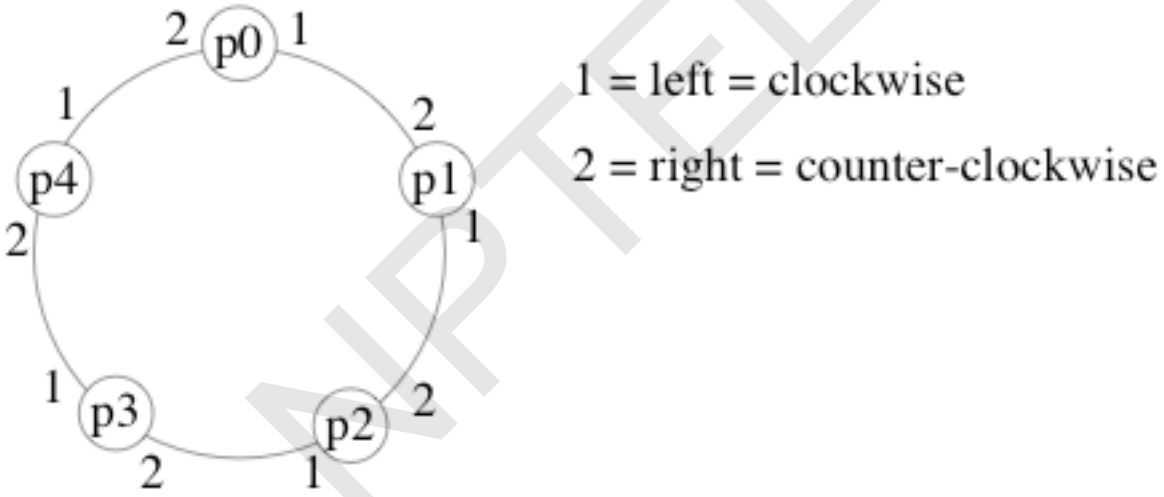
Uses of Leader Election

- A leader can be used to coordinate activities of the system:
  - find a ***spanning tree*** using the leader as the root
  - reconstruct a ***lost token*** in a token-ring network

- In this lecture, we will study the leader election in rings.

Definition: (1)Ring Networks

- In an **oriented ring**, processors have a consistent notion of left and right



- For example, if messages are always forwarded on channel 1, they will cycle clockwise around the ring

Definition: (2) Anonymous Rings

- How to model situation when **processors do not have unique identifiers?**

- **First attempt:** require each processor to have the same state machine

Definition: (3) Uniform (Anonymous) Algorithms

- A **uniform** algorithm **does not use the ring size** (same algorithm for each size ring)
  - Formally, every processor in every size ring is modeled with the same state machine

- A **non-uniform** algorithm **uses the ring size** (different algorithm for each size ring)
  - Formally, for each value of n, every processor in a ring of size n is modeled with the same state machine A_n .

- Note the lack of unique ids.

Impossibility: Leader Election in Anonymous Rings

Theorem: There is ***no*** **leader election algorithm for anonymous rings,** even if

algorithm knows the ring size (non-uniform) and synchronous model

Proof Sketch:

- Every processor begins in same state with same outgoing messages (since anonymous)
- Every processor receives same messages, does same state transition, and sends same messages in round 1
- Ditto for rounds 2, 3, …
- Eventually some processor is supposed to enter an elected state. But then they all would.

Leader Election in Anonymous Rings

- Proof sketch shows that either safety (never elect more than one leader) or liveness (eventually elect at least one leader) is violated.

- Since the theorem was proved for non-uniform and synchronous rings, the same result holds for weaker (less well-behaved) models:
  - uniform
  - asynchronous

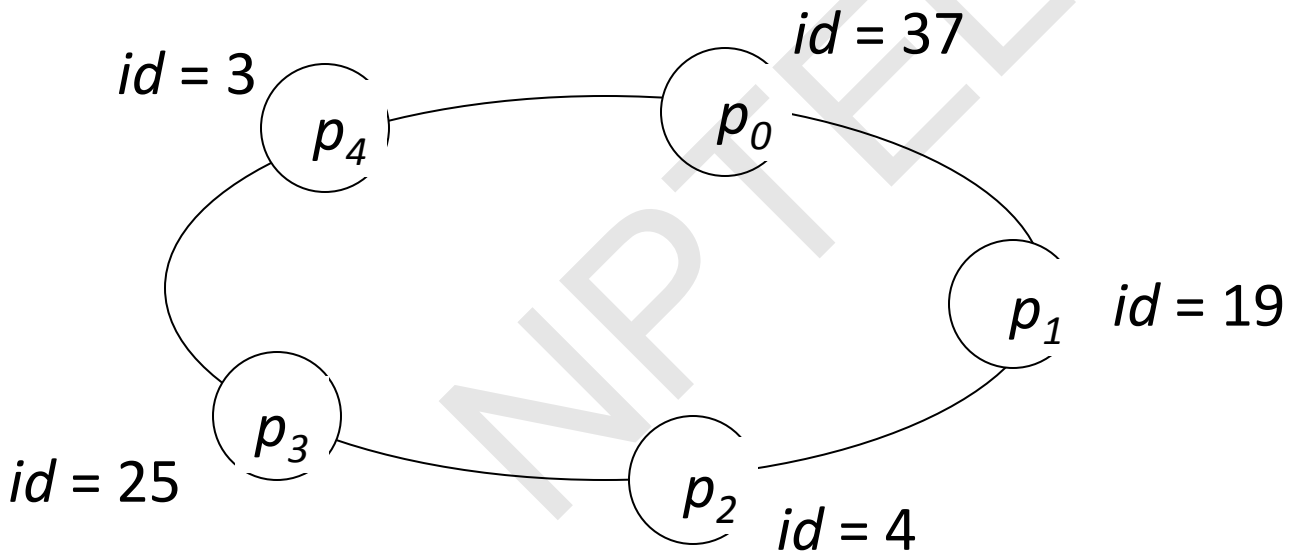
Rings with Identifiers

- Assume each processor has a unique id.

- Don't confuse indices and ids:
  - **indices** are 0 to *n - 1*; used only for analysis, not available to the processors
  - **ids** are arbitrary nonnegative integers; are available to the processors through local variable *id.*

Specifying a Ring

- Start with the smallest id and list ids in clockwise order.



- Example: 3, 37, 19, 4, 25

Uniform (Non-anonymous) Algorithms

- **Uniform** algorithm: there is one state machine for every id, no matter what size ring
- **Non-uniform** algorithm: there is one state machine for every id and every different ring size
- These definitions are tailored for leader election in a ring.

O(n^2) Messages LE Algorithm: LeLann-Chang-Roberts (LCR) algorithm

- send value of own *id* to the left
- when receive an id *j* (from the right):
  - if *j* > *id* then
    - forward *j* to the left (this processor has lost)
  - if *j* = *id* then
    - elect self (this processor has won)
  - if *j* < *id* then
    - do nothing

Analysis of $O(n^2)$ Algorithm

Correctness: Elects processor with largest id.

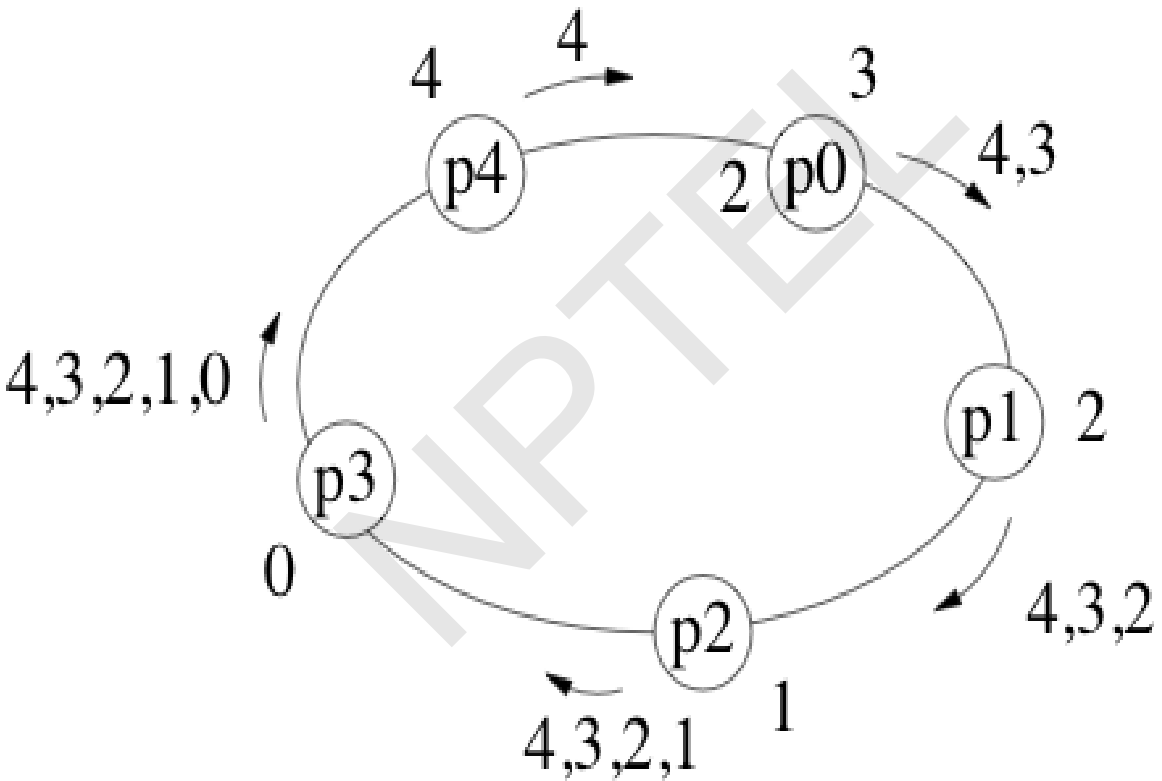
- message containing largest id passes through every processor

Time: $O(n)$

Message complexity: Depends how the ids are arranged.

- largest id travels all around the ring (n messages)
- 2nd largest id travels until reaching largest
- 3rd largest id travels until reaching largest or second largest etc.

Analysis of $O(n^2)$ Algorithm



Analysis of $O(n^2)$ Algorithm

- Worst way to arrange the ids is in decreasing order:
  - 2nd largest causes $n - 1$ messages
  - 3rd largest causes $n - 2$ messages etc.

- **Total number of messages is:**

$$n + (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$$

Analysis of $O(n^2)$ Algorithm

- Clearly, the algorithm never sends more than $O(n^2)$ messages in any admissible execution. Moreover, there is an admissible execution in which the algorithm sends $\Theta(n^2)$ messages; Consider the ring where the identifiers of the processor are *0,……, n-1* and they are ordered as in Figure 3.2. In this configuration, the message of processor with identifier *i* is send exactly *i+1* times, Thus the total number of messages, including the *n* termination messages, is

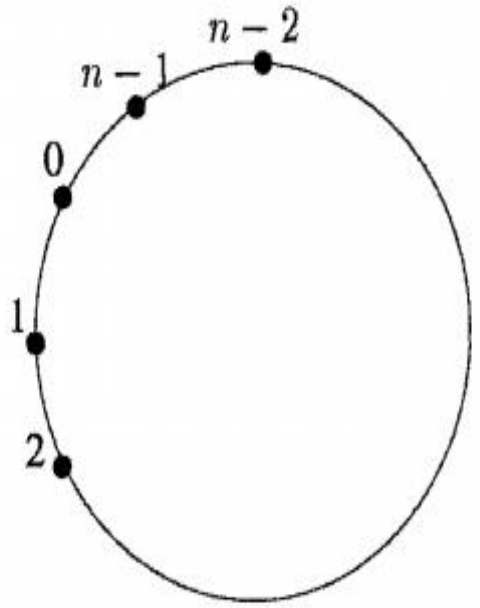


Fig. 3.2 Ring with $\Theta(n^2)$ messages.

$$n + \sum_{i=0}^{n-1} (i+1) = \Theta(n^2)$$

Clockwise Unidirectional Ring

Can We Use Fewer Messages?

- The $O(n^2)$ algorithm is simple and works in both synchronous and asynchronous model.
- But can we solve the problem with fewer messages?

Idea:

- Try to have messages containing smaller ids travel smaller distance in the ring

O(*nlogn*) Messages LE Algorithm: The Hirschberg and Sinclair (HS) algorithm

- To describe the algorithm, we first define the ***k-neighbourhood*** of a **processor p_i** in the ring to be the set of processors that are at **distance at most *k* from p_i in the ring** (either to the left or to the right). Note that the ***k-neighbourhood*** of a processor includes exactly ***2k+1*** processors.
- The algorithm operates in phases; it is convenient to start numbering the phases with 0. In the ***kth* phase** a processor tries to become a ***winner*** for that phase; to be a winner, it must have the **largest id** in its ***2^k-neighborhood***. Only processors that are winners in the **kth phase** continue to compete in the ***(k+1)*-st phase**, Thus fewer processors proceed to higher phases, until at the end, only one processor is a winner and it is elected as the leader of the whole ring.

The HS Algorithm: Sending Messages

Phase 0

- In more detail, in phase 0, each processor attempts to become a phase 0 winner and sends a **<probe>** message containing its identifier to its ***1-neighborhood***, that is, to each of its two neighbors.
- If the identifier of the neighbor receiving the probe is greater than the identifier in the probe, it swallows the probe; otherwise, it sends back a **<reply>** message.
- If a processor receives a reply from both its neighbors, then the processor becomes a phase 0 winner and continues to phase 1.

The HS Algorithm: Sending Messages

Phase k

- In general, in phase k, a processor p_i that is a phase ***k-1*** **winner** sends **<probe>** messages with its identifier to its ***2^k-neighborhood*** (one in each direction). Each such message traverses ***2^k processors*** one by one, A probe is swallowed by a processor if it contains an identifier that is smaller than its own identifier.
- If the probe arrives at the last processor on the neighbourhood without being swallowed, then that last processor sends back a **<reply>** message to p_i. If p_i receives replies from both directions, it becomes a phase k winner, and it continues to phase ***k+1***. A processor that receives its own **<probe>** message terminates the algorithm as the leader and sends a termination message around the ring.

Algorithm 5 Asynchronous leader election: code for processor p_i, $0 \leq i < n$.

Initially, *asleep* = true

1: upon receiving no message:
2: if *asleep* then
3: *asleep* := false
4: send $\langle$probe,*id*,0,1$\rangle$ to left and right

5: upon receiving $\langle$probe,j,k,$d\rangle$ from left (resp., right):
6: if $j = id$ then terminate as the leader
7: if $j > id$ and $d < 2^k$ then // forward the message
8: send $\langle$probe,j,k,$d+1\rangle$ to right (resp., left) // increment hop counter
9: if $j > id$ and $d \geq 2^k$ then // reply to the message
10: send $\langle$reply,j,$k\rangle$ to left (resp., right)
// if $j < id$, message is swallowed

11: upon receiving $\langle$reply,j,$k\rangle$ from left (resp., right):
12: if $j \neq id$ then send $\langle$reply,j,$k\rangle$ to right (resp., left) // forward the reply
13: else // reply is for own probe
14: if already received $\langle$reply,j,$k\rangle$ from right (resp., left) then
15: send $\langle$probe,*id*,$k+1$,1$\rangle$ // phase k winner

The HS Algorithm

- The pseudocode appears in **Algorithm 5**. Phase *k* for a processor corresponds to the period between its sending of a **<probe>** message in line 4 or 15 with third parameter k and its sending of a **<probe>** message in line 4 or 15 with third parameter ***k+1***. The details of sending the termination message around the ring have been left out in the code, and only the leader terminates.

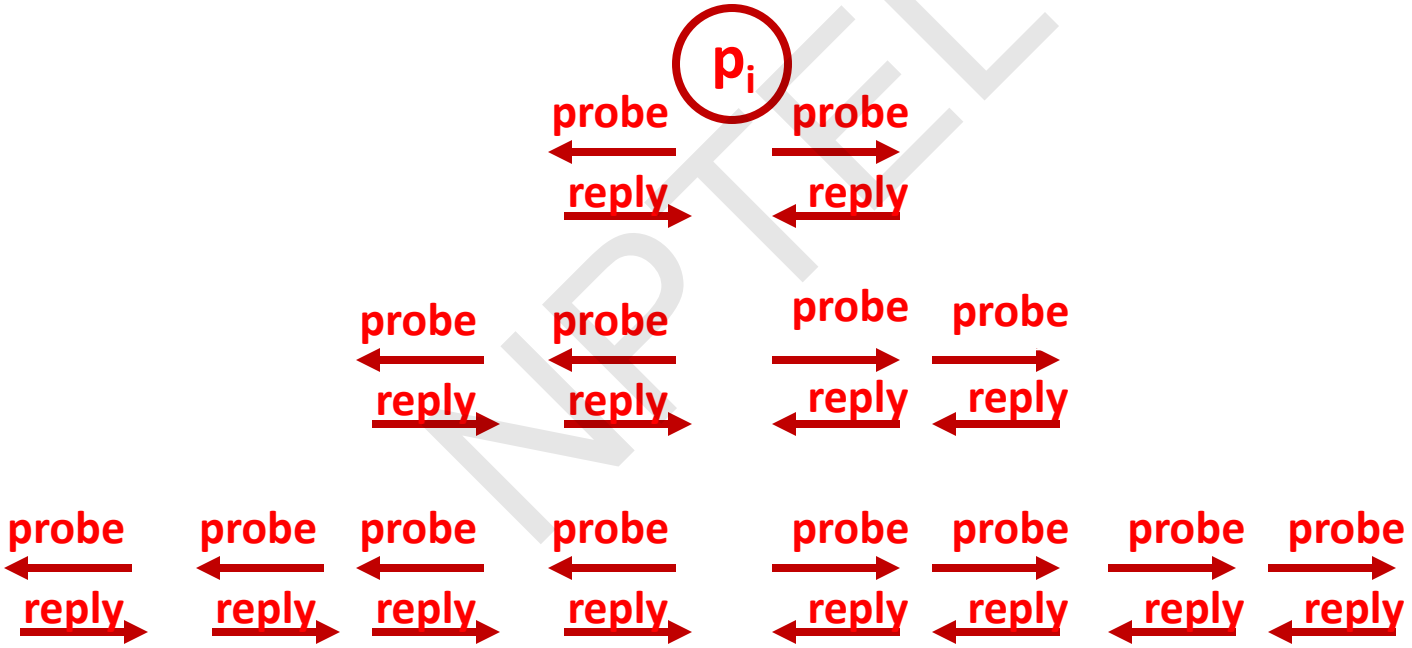
- The correctness of the algorithm follows in the same manner as in the simple algorithm, because they have the same swallowing rules.

- It is clear that the probes of the processor with the maximal identifier are never swallowed; therefore, this processor will terminate the algorithm as a leader. On the other hand, it is also clear that no other **<probe>** can traverse the whole ring without being swallowed. Therefore, the processor with the maximal identifier is the only leader elected by the algorithm.

O(n log *n)* Leader Election Algorithm

- Each processor tries to probe successively larger neighborhoods in both directions
  - size of neighborhood ***doubles*** in each phase
- If probe reaches a node with a larger id, the probe stops
- If probe reaches end of its neighborhood, then a reply is sent back to initiator
- If initiator gets back replies from both directions, then go to next phase
- If processor receives a probe with its own id, it elects itself

O(n log *n)* Leader Election Algorithm



Analysis of $O(n \log n)$ Leader Election Algorithm

Correctness:

- Similar to $O(n^2)$ algorithm.

Message Complexity:

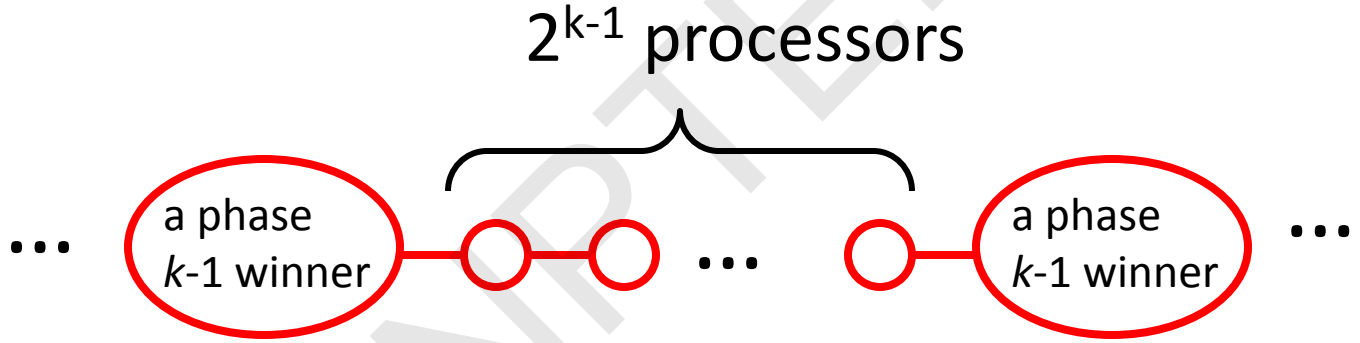
- Each message belongs to a particular phase and is initiated by a particular processor
- Probe distance in phase k is 2^k
- Number of messages initiated by a processor in phase k is at most $4*2^k$ (probes and replies in both directions)

Analysis of *O(n* log *n)* Leader Election Algorithm

- How many processors initiate probes in phase *k* ?
- For *k = 0*, every processor does
- For *k > 0*, every processor that is a "winner" in phase *k - 1* does
  - "winner" means has largest id in its 2^{k-1} neighborhood

Analysis of *O(n log n)* Leader Election Algorithm

- Maximum number of phase *k - 1* winners occurs when they are packed as densely as possible:



- **Total number of phase *k - 1* winners is at most**

  $n/(2^{k-1} + 1)$

Analysis of $O(n \log n)$ Leader Election Algorithm

- How many phases are there?
- At each phase the number of (phase) winners is cut approx. in half
  - **from $n/(2^{k-1} + 1)$ to $n/(2^k + 1)$**
- So after approx. $\log_2 n$ phases, only one winner is left.
  - more precisely, **max phase is $\lceil \log(n-1) \rceil + 1$**

Analysis of $O(n \log n)$ Leader Election Algorithm

- Total number of messages is sum, over all phases, of number of winners at that phase times number of messages originated by that winner:

$$\leq 4n + n + \sum_{k=1}^{\lceil \log(n-1) \rceil + 1} 4 \bullet 2^k \bullet n/(2^{k-1}+1)$$

$$< 8n(\log n + 2) + 5n$$

$$= O(n \log n)$$

phase 0 msgs

termination msgs

msgs for phases 1 to $\lceil \log(n-1) \rceil + 1$

Can We Do Better?

- The $O(n \log n)$ algorithm is more complicated than the $O(n^2)$ algorithm but uses fewer messages in worst case.
- Works in both synchronous and asynchronous case.
- Can we reduce the number of messages even more?
- Not in the asynchronous model…

Lower bound for LE algorithm

But, can we do better than O(n log n)?

Theorem: Any leader election algorithm for asynchronous rings whose size is not known a priori has Ω(n log n) message complexity (holds also for unidirectional rings).

- Both LCR and HS are comparison-based algorithms, i.e. they use the identifiers only for comparisons (<; >;=).
- In synchronous networks, O(n) message complexity can be achieved if general arithmetic operations are permitted (non-comparison based) and if time complexity is unbounded.

Overview of LE in Rings with Ids

- There exist algorithms when nodes have unique ids.
- We have evaluated them according to their ***message complexity***.
- **Asynchronous ring:**
  - $\Theta(n \log n)$ messages
- **Synchronous ring:**
  - $\Theta(n)$ messages under certain conditions
  - otherwise $\Theta(n \log n)$ messages
- All bounds are asymptotically tight.

Conclusion

- This lecture provided an in-depth study of the leader election problem in message-passing systems for a ring topology.

- We have presented the different algorithms for leader election problem by taking the cases like **anonymous/non-anonymous rings, uniform/non-uniform rings and synchronous/ asynchronous rings**

Leader Election
(Ring LE & Bully LE Algorithm)

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the underlying concepts of **'leader election problem'**, which has been very useful for many variant of distributed systems including today's Cloud Computing Systems.

- Then we will present the different '**classical algorithms for leader election problem' i.e. Ring LE and Bully LE algorithms** and also discuss how election is done in some of the popular systems in Industry such as **Google's Chubby** and **Apache's Zookeeper system**.

Need of Election

- **Example 1:** Suppose 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

Some more motivating examples

- **Example 2:** Group of NTP servers: who is the root server?

- **Other systems that need leader election:**

  Apache Zookeeper, Google's Chubby

- Leader is useful for coordination among distributed servers

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!)
  - Then what?

- **Goals of Election algorithm:**
  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 id.
- Messages are eventually delivered.
- Failures may occur during the election protocol.

Calling for an Election

- Any process can **call** for an **election**.
- A process can call for **at most one election at a time**.
- Multiple processes are allowed to call an election simultaneously.
  - **All of them together must yield only a single leader**
- The result of an election should not depend on which process calls for it.

Formally: Election Problem

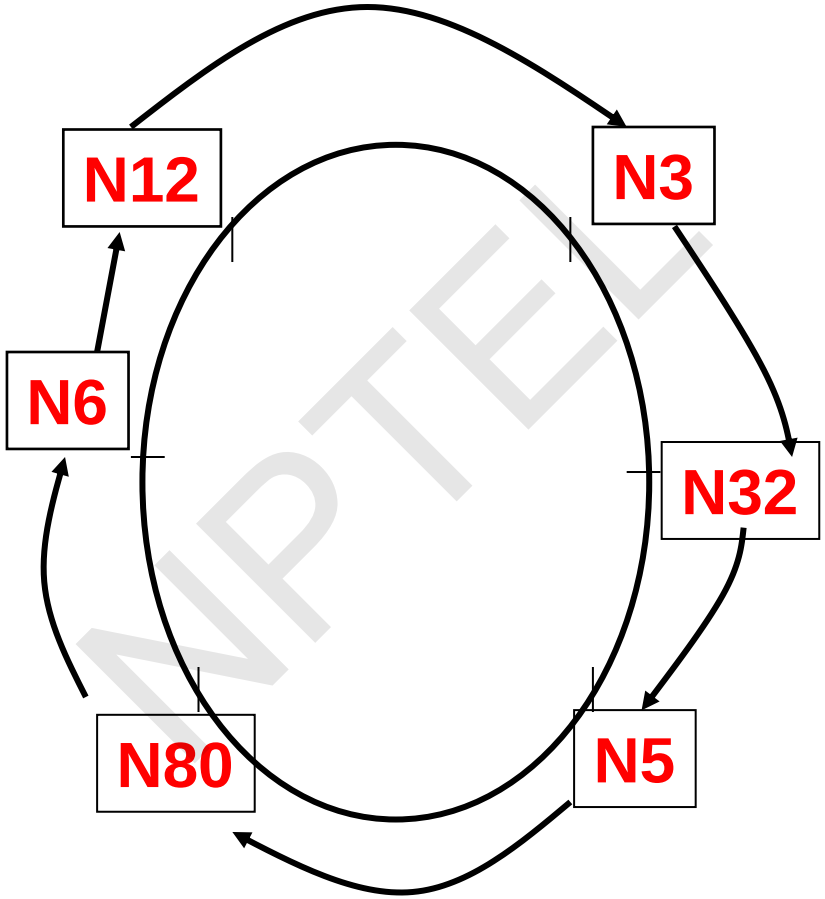
- A run of the election algorithm must always guarantee at the end:
  - **Safety**: For all non-faulty processes *p*: (*p*'s elected = (q: a particular non-faulty process with the best attribute value) or Null)
  - **Liveness**: For all election runs: (election run terminates) & for all non-faulty processes *p*: *p*'s elected is not Null
- At the end of the election protocol, the non-faulty process with the **best (highest)** election attribute value is elected.
  - Common attribute : leader has highest id
  - Other attribute examples: leader has highest IP address, or fastest cpu, or most disk space, or most number of files, etc.

(i) Classical Algorithm: Ring Election

The Ring:

- *N* processes are organized in a logical ring
  - Similar to **ring in Chord p2p system**
  - *i*-th process p_i has a communication channel to $p_{(i+1) \bmod N}$
  - All messages are sent clockwise around the ring.

The Ring



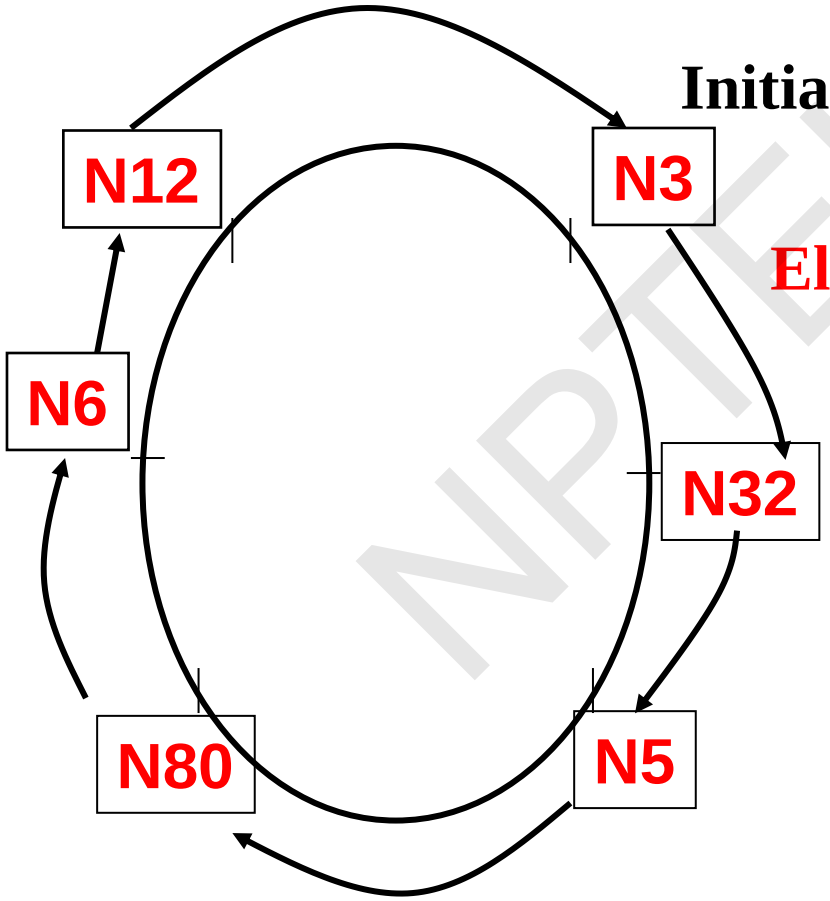
The Ring Election Protocol

- Any process p_i that discovers the old coordinator has failed initiates an **"Election"** message that contains p_i's own id:attr. (attr: attribute) This is the *initiator* of the election.
- When a process p_i receives an "Election" message, it compares the attr in the message with its own attr.
  - If the arrived attr is greater, p_i forwards the message.
  - If the arrived attr is smaller and p_i has not forwarded an election message earlier, it overwrites the message with its own id:attr, and forwards it.
  - If the arrived id:attr matches that of p_i, then p_i's attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an **"Elected"** message to its neighbor with its id, announcing the election result.

The Ring Election Protocol (2)

- When a process p_i receives an **"Elected"** message, it
  - sets its variable $elected_i \leftarrow$ id of the message.
  - forwards the message unless it is the new coordinator.

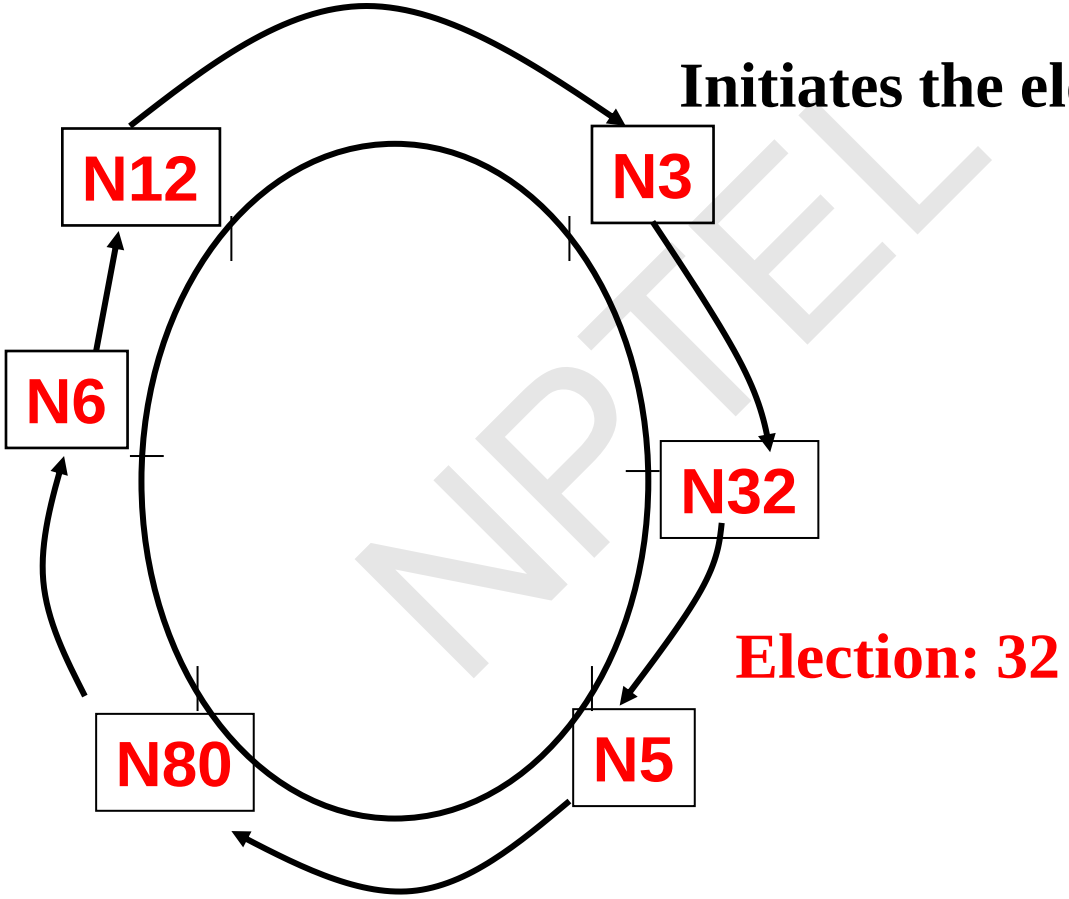
Ring Election: Example



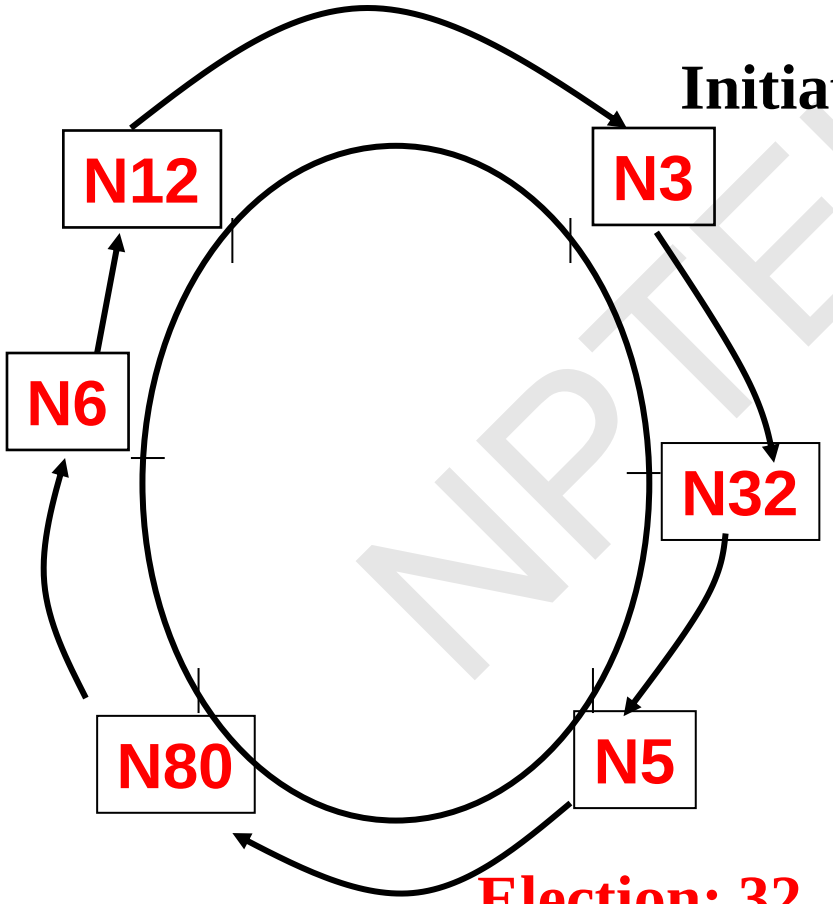
Initiates the election

Election: 3

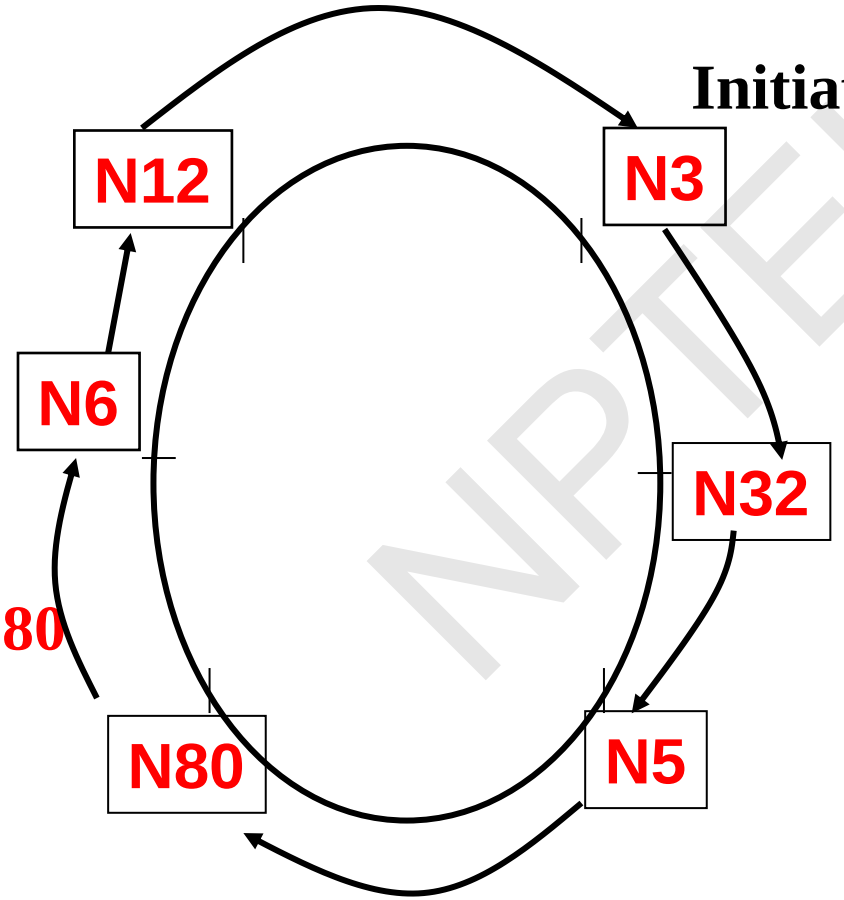
Goal: Elect highest id process as leader



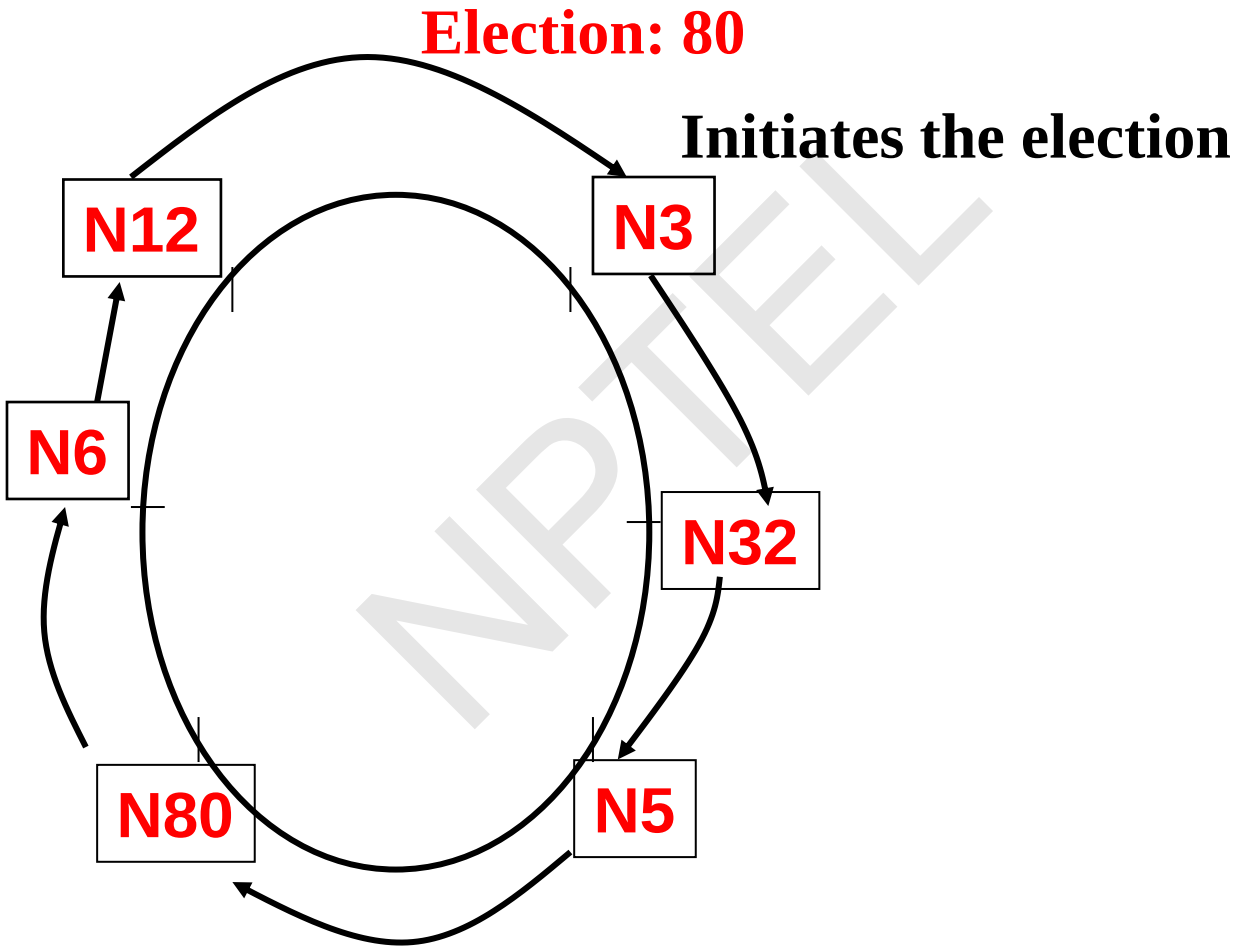
Goal: Elect highest id process as leader



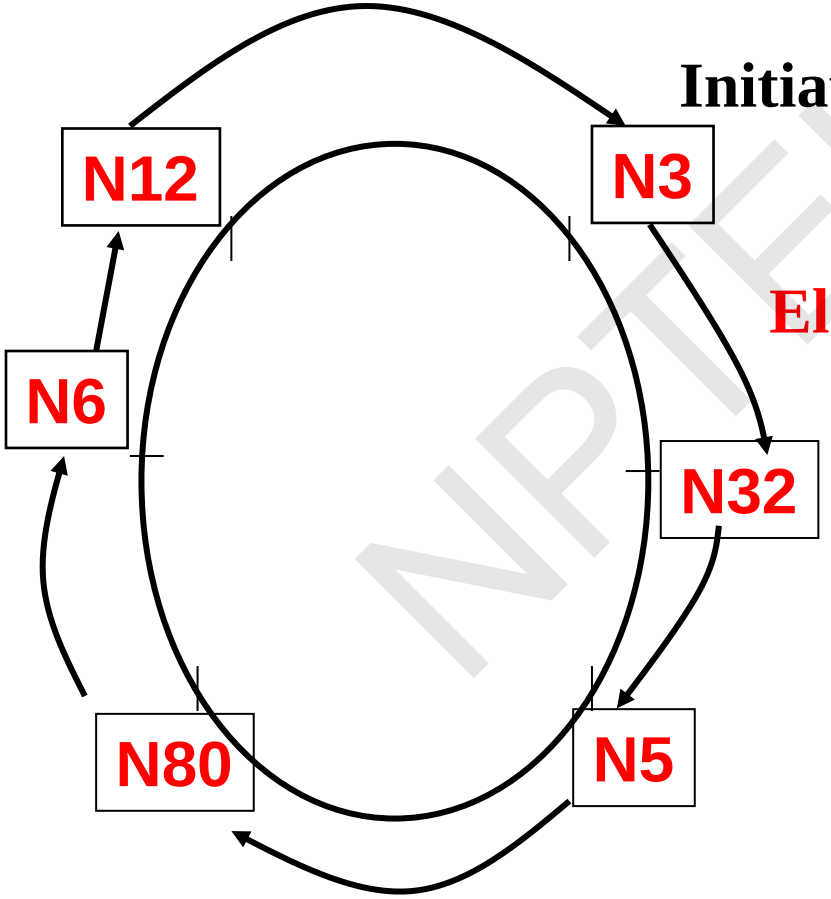
Goal: Elect highest id process as leader



Goal: Elect highest id process as leader



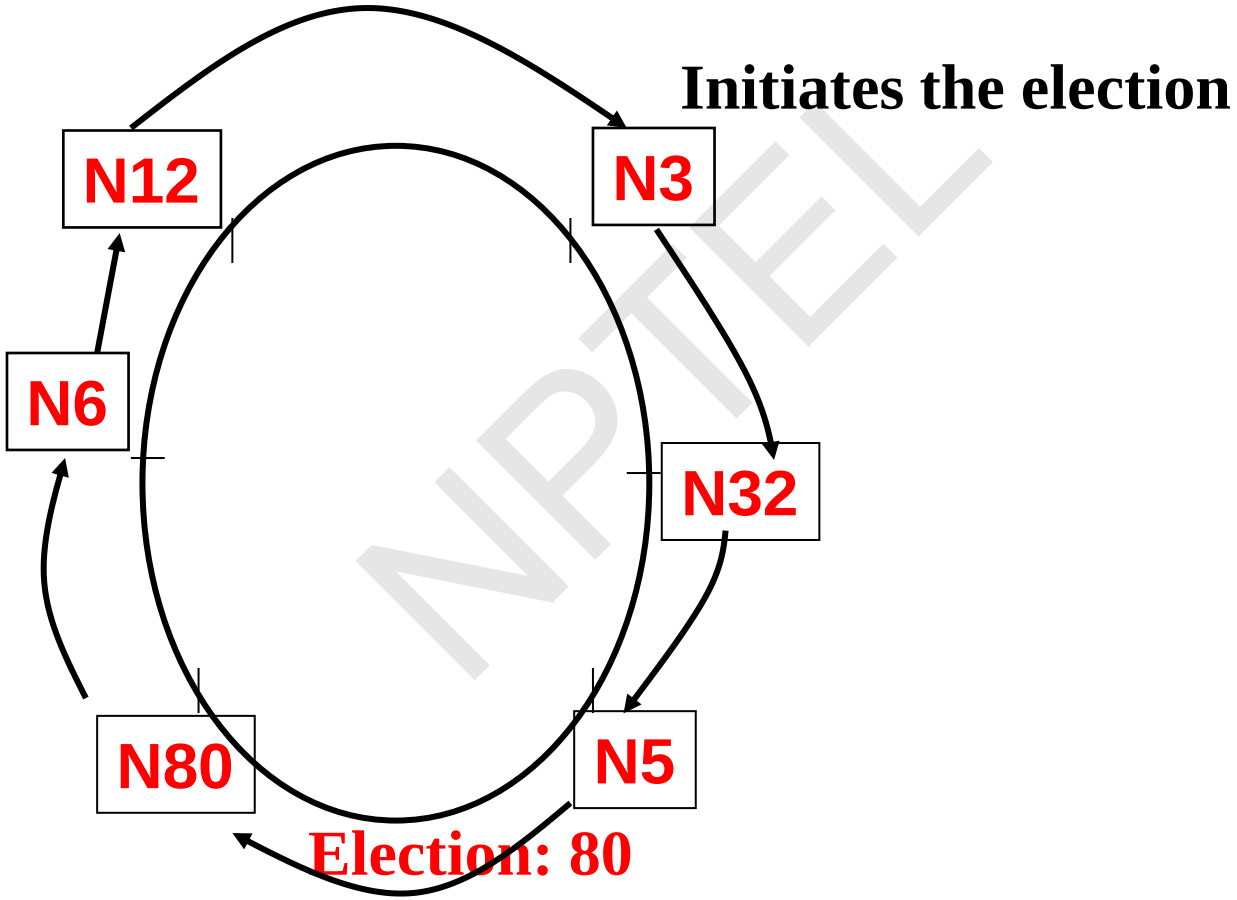
Goal: Elect highest id process as leader



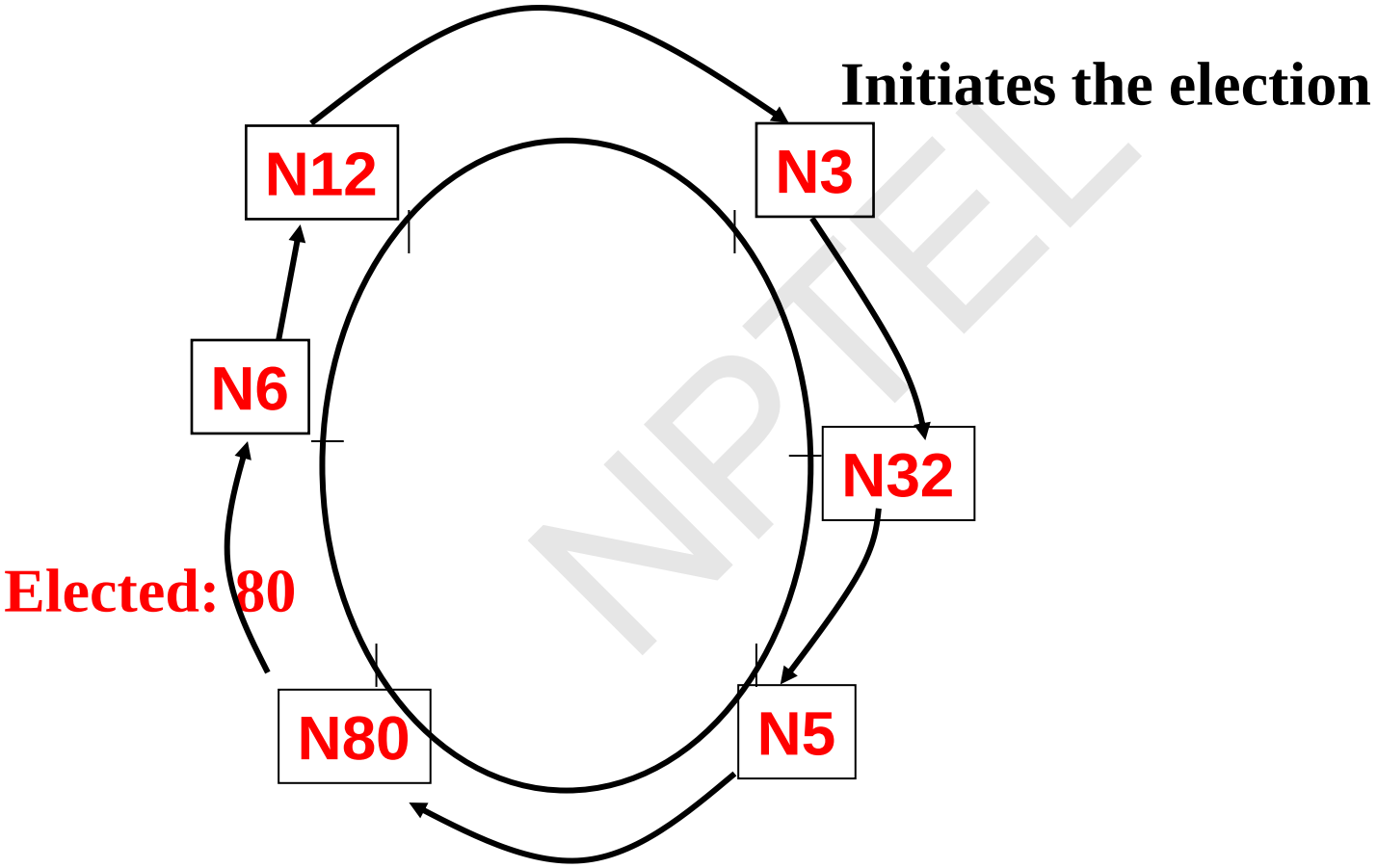
Initiates the election

Election: 80

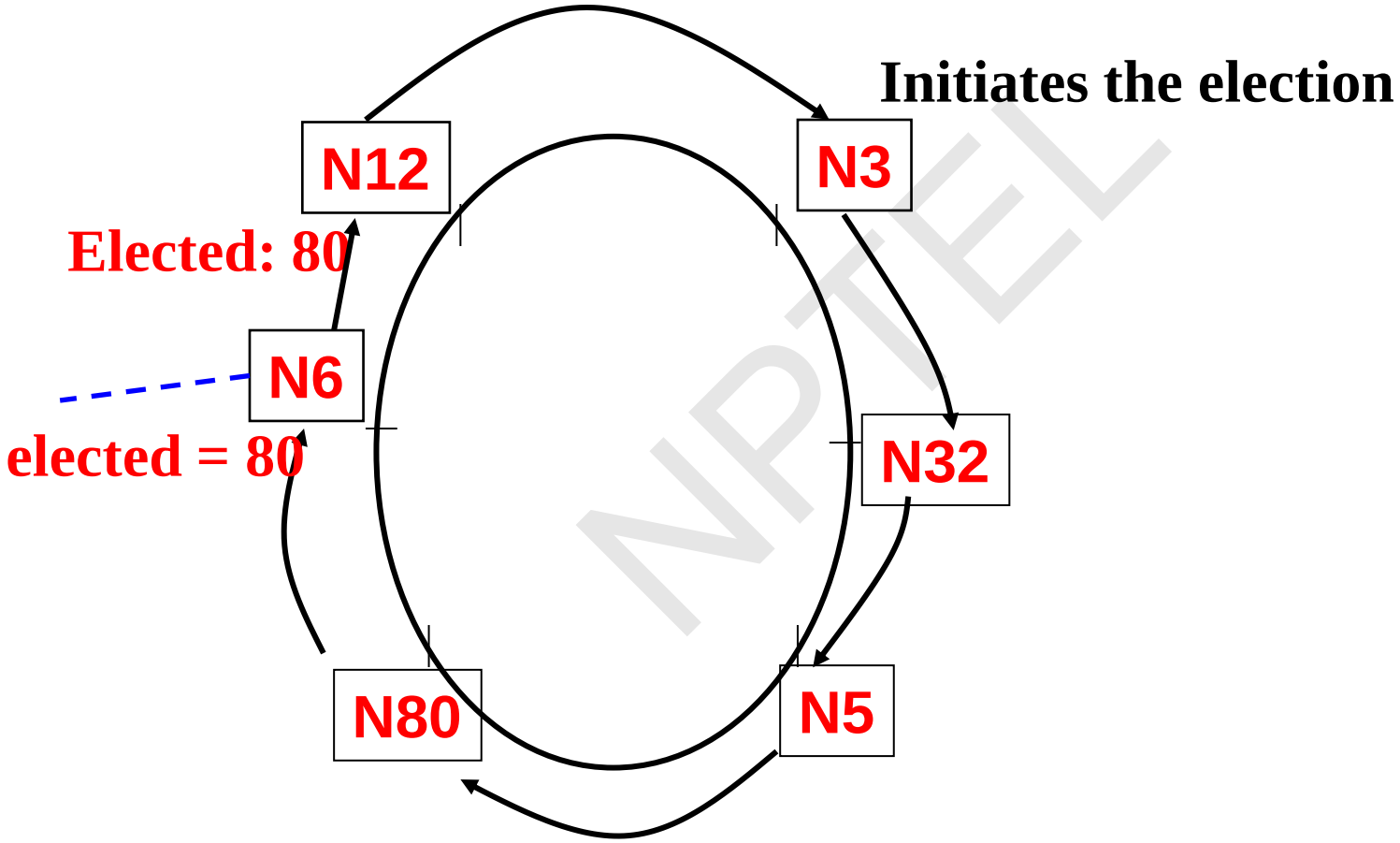
Goal: Elect highest id process as leader



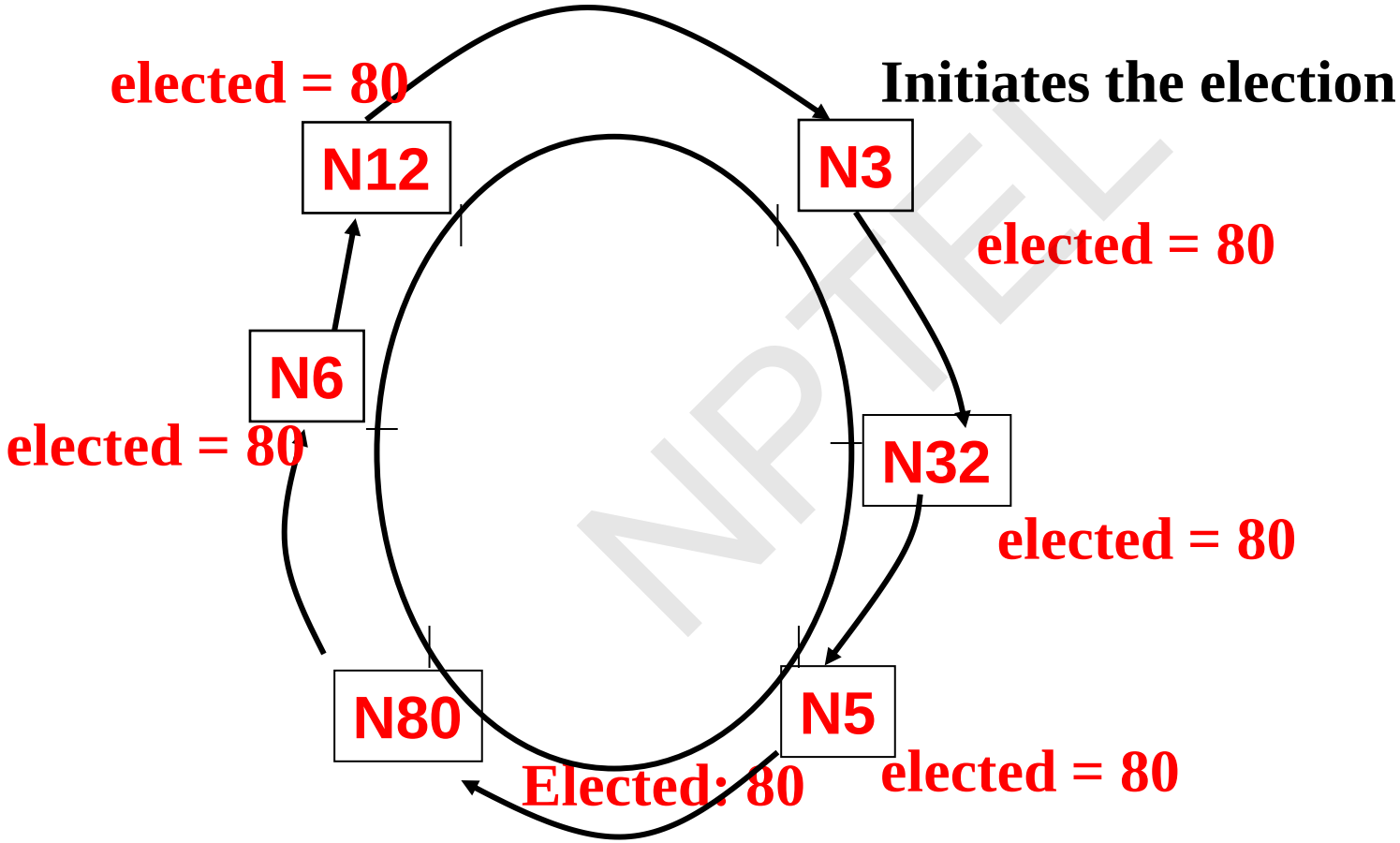
Goal: Elect highest id process as leader



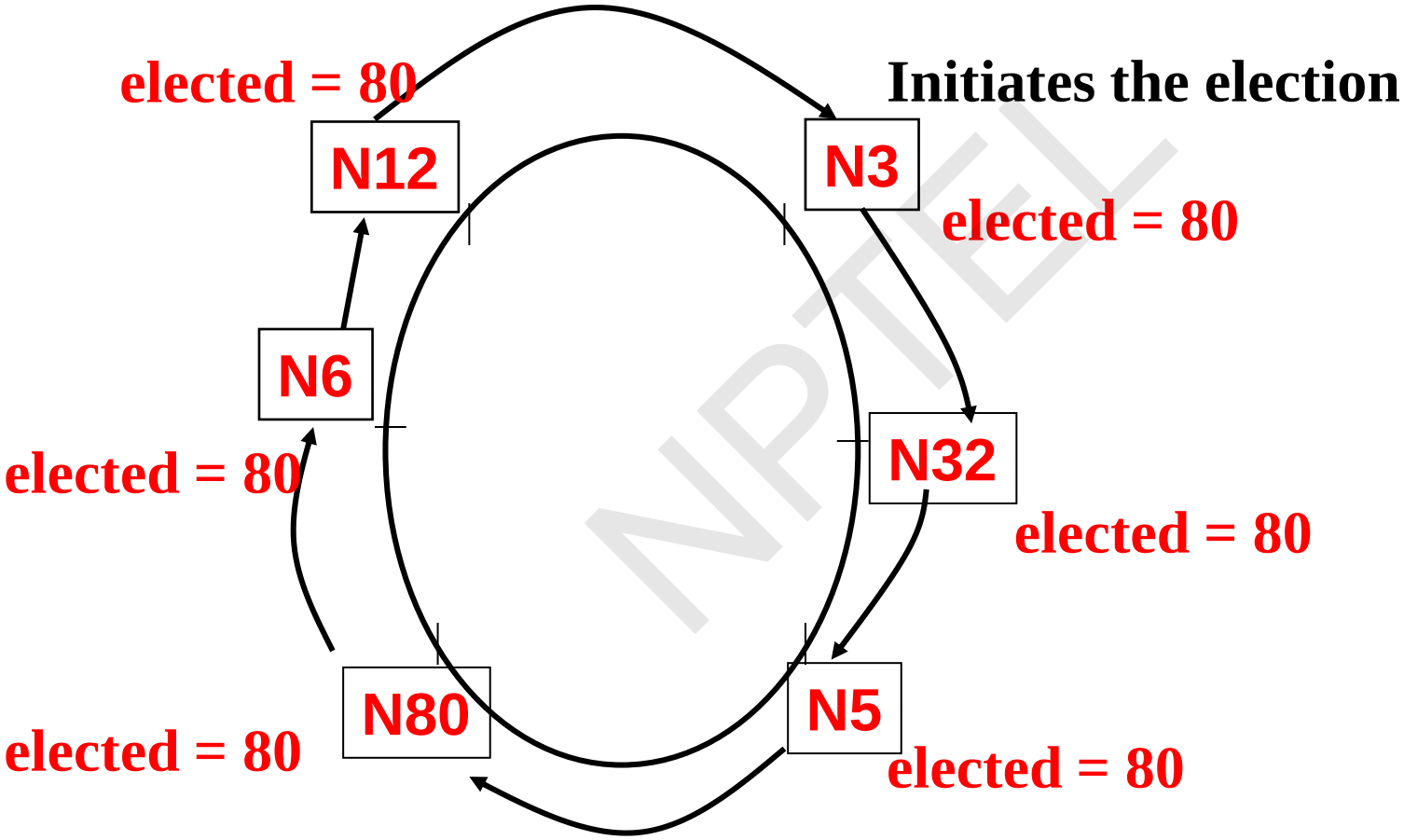
Goal: Elect highest id process as leader



Goal: Elect highest id process as leader



Goal: Elect highest id process as leader



Goal: Elect highest id process as leader

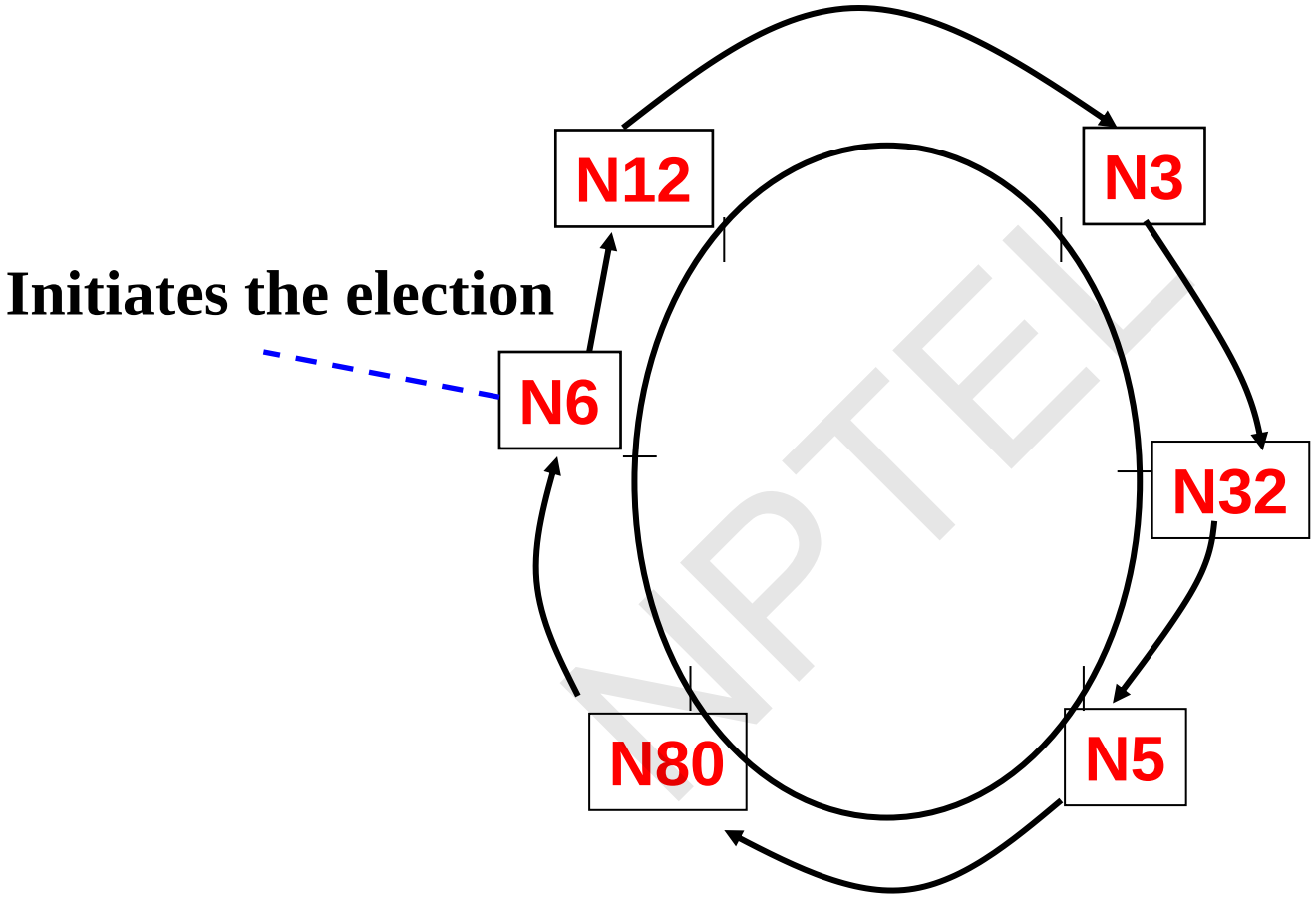
Analysis

- Let's assume no failures occur during the election protocol itself, and there are *N* processes

- **How many messages?**

- Worst case occurs when the initiator is the ring successor of the would-be leader

Worst-case



Goal: Elect highest id process as leader

Worst-case Analysis

- **(*N-1*) messages** for Election message to get from Initiator (N6) to would-be coordinator (N80)
- ***N* messages** for Election message to circulate around ring without message being changed
- ***N* messages** for Elected message to circulate around the ring

- **Message complexity: (*3N-1*) messages**
- **Completion time: (*3N-1*) message transmission times**
- Thus, if there are no failures, election terminates **(liveness)** and everyone knows about highest-attribute process as leader **(safety)**

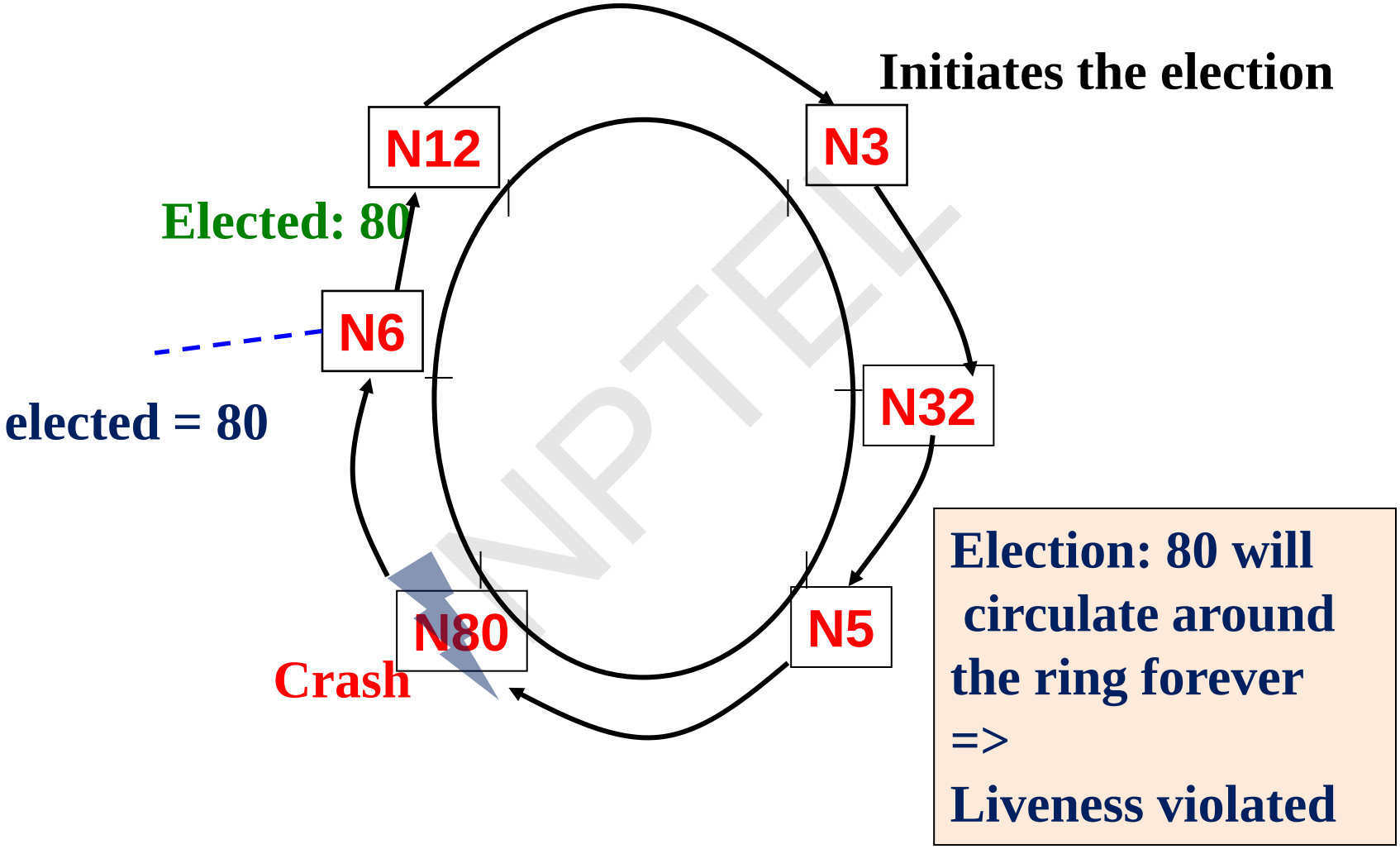
Best Case?

- Initiator is the would-be leader, i.e., N80 is the initiator
- **Message complexity:** *2N* messages
- **Completion time:** *2N* message transmission times

Multiple Initiators?

- 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



Fixing for failures: First Option

- **First 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: Second Option

- **Second option:** use the failure detector (FD)
- 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
  - **Incompleteness in FD** => N80's failure might be missed => Violation of Safety
  - **Inaccuracy in FD** => N80 mistakenly detected as failed
    - => new election runs initiated forever
    - => Violation of Liveness

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, so is leader election!**
- Consensus-like protocols such as Paxos used in industry systems for leader election

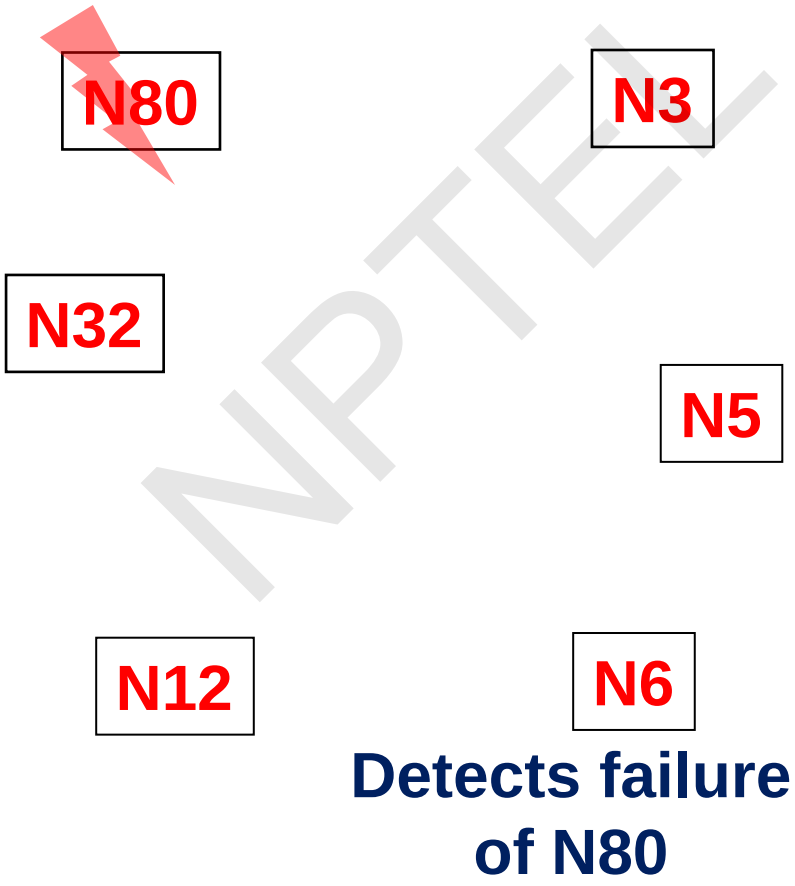
(ii) Classical Algorithm: Bully Algorithm

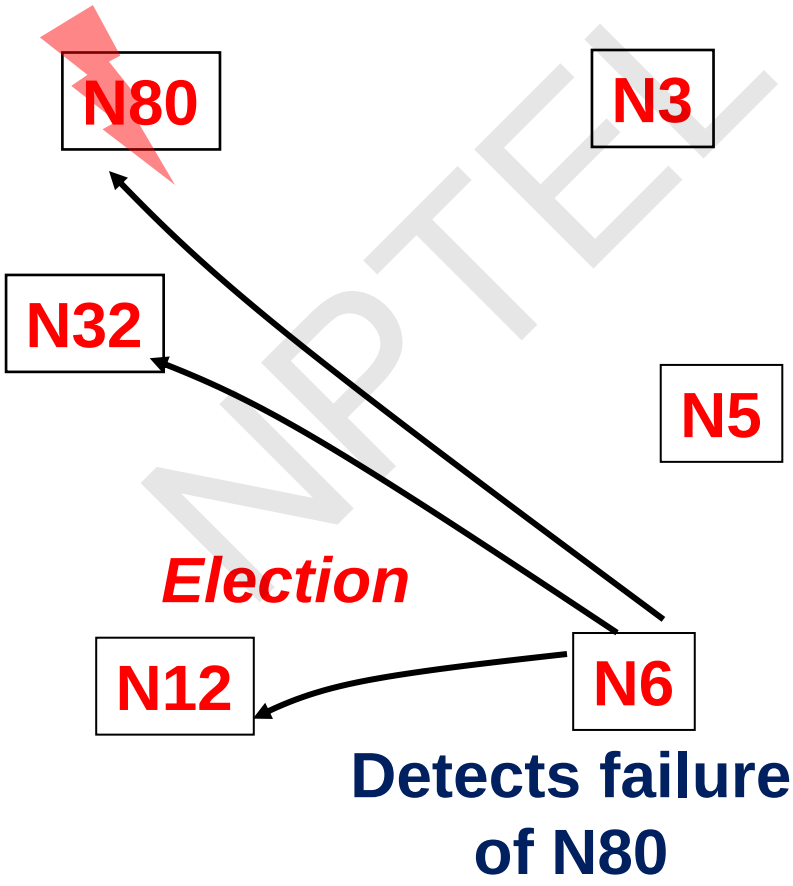
- All processes know other process' ids
- When a process finds the coordinator has failed (via the failure detector):
  - **if** it knows its id is the highest
    - it elects itself as coordinator, then sends a ***Coordinator*** message to all processes with lower identifiers. Election is completed.
  - **else**
    - it initiates an election by sending an ***Election*** message
    - (contd…)

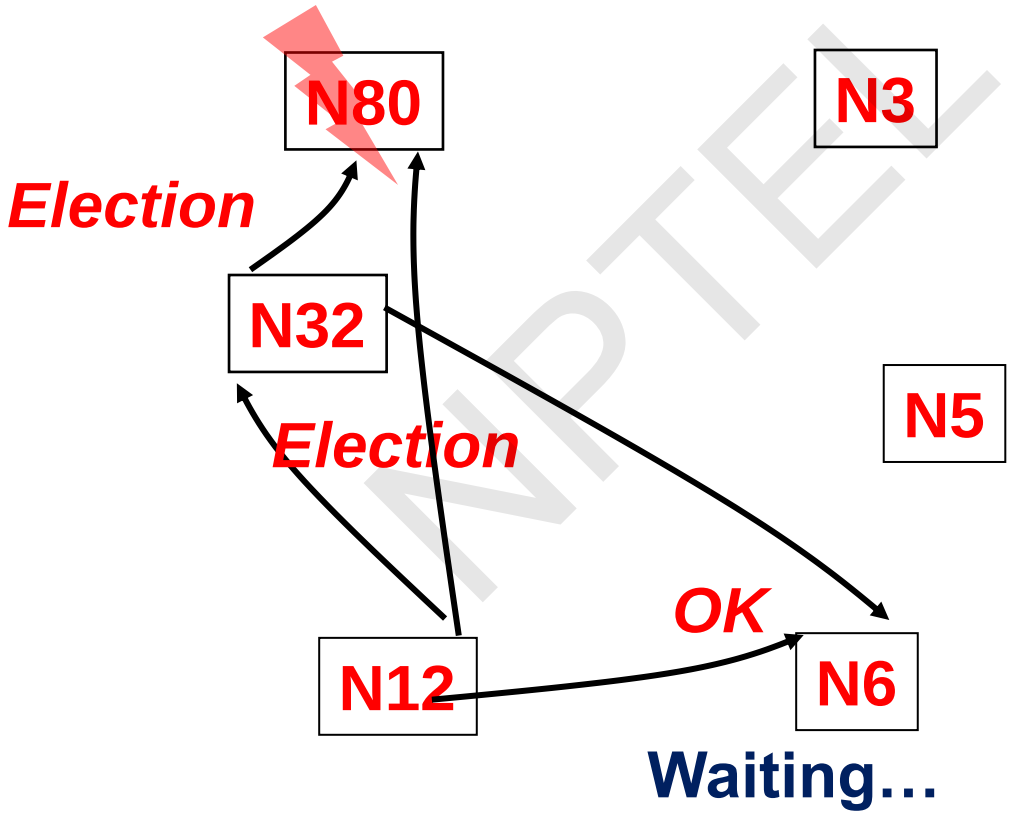
Bully Algorithm (2)

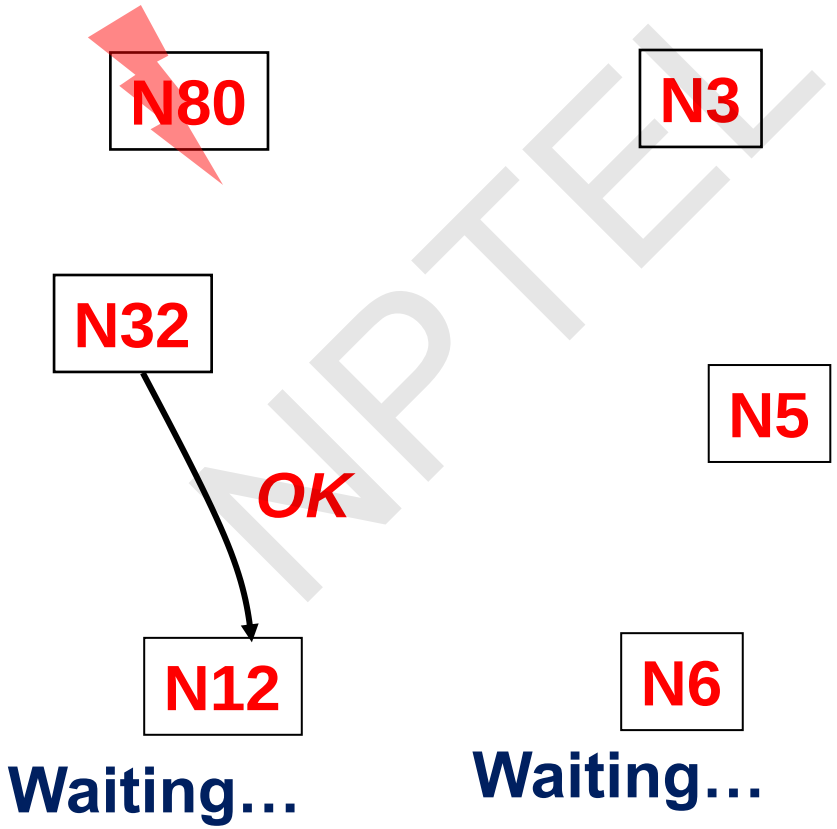
- **else** it initiates an election by sending an ***Election*** message
  - Sends it to only processes that have a *higher id than itself*.
  - **if** receives no answer within timeout, calls itself leader and sends ***Coordinator*** message to all lower id processes. Election completed.
  - **if** an answer received however, then there is some non-faulty higher process => so, wait for coordinator message. If none received after another timeout, start a new election run.
- A process that receives an ***Election*** message replies with ***OK*** message, and starts its own leader election protocol (unless it has already done so)

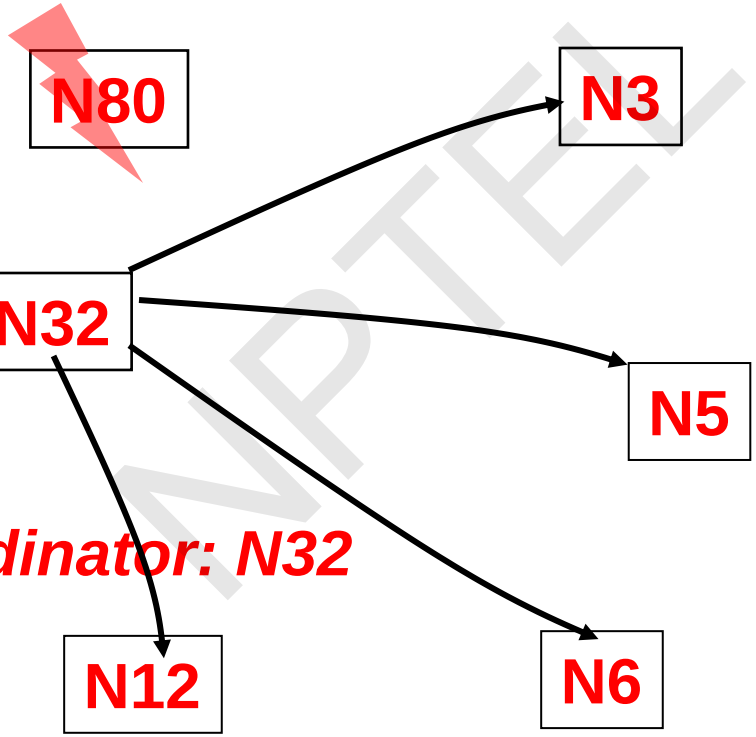
Bully Algorithm: Example




N80
N3
N32
N5
Election
N12
N6
Detects failure
of N80


N80
N3
Election
N32
N5
Election
OK
N12
N6
Waiting…



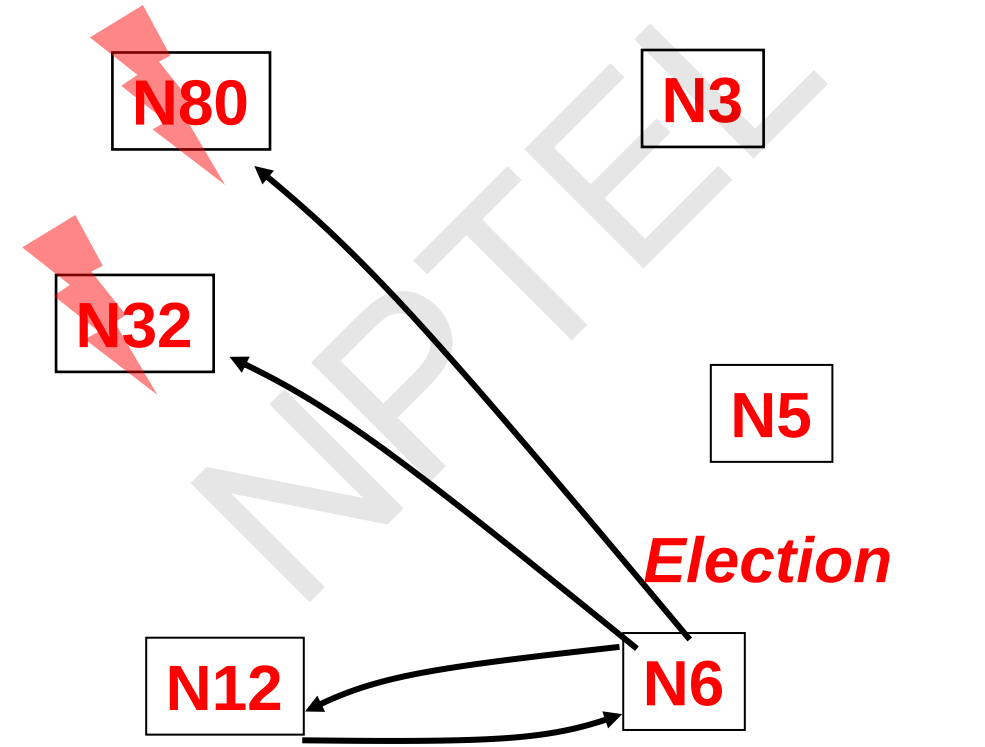


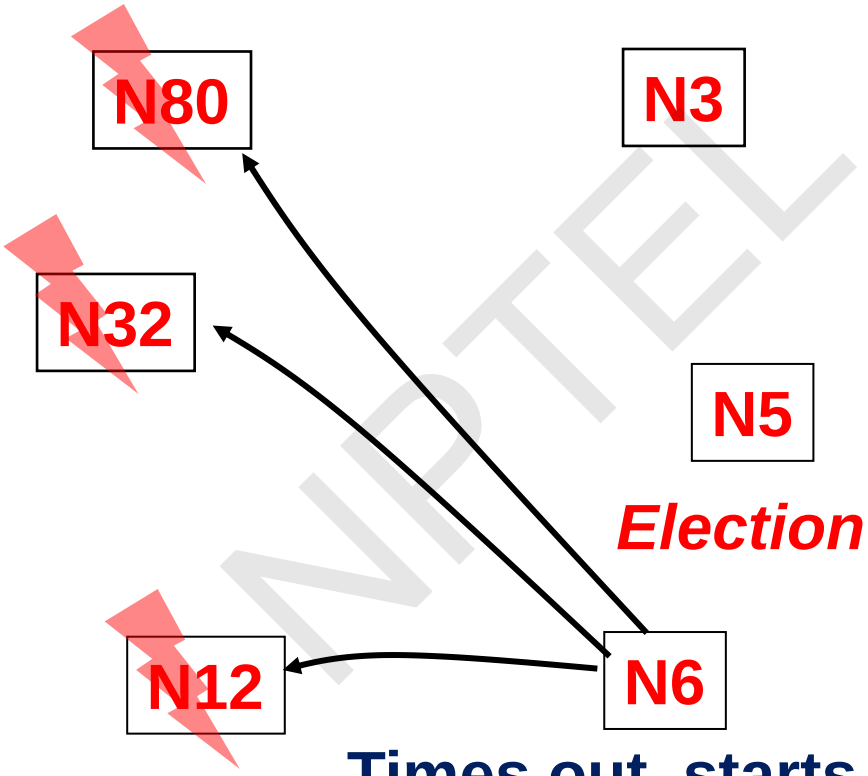
Times out waiting for N80's response

Coordinator: N32

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

Failures and Timeouts

- If failures stop, eventually will elect a leader
- How do you set the timeouts?
- Based on Worst-case time to complete election
  - 5 message transmission times if there are no failures during the run:

1. **Election from lowest id server in group**
2. **Answer to lowest id server from 2nd highest id process**
3. **Election from 2nd highest id server to highest id**
4. **Timeout for answers @ 2nd highest id server**
5. **Coordinator from 2nd highest id server**

Analysis

- **Worst-case completion time:** 5 message transmission times
  - When the process with the lowest id in the system detects the failure.
    - (*N-1*) processes altogether begin elections, each sending messages to processes with higher ids.
    - *i*-th highest id process sends (*i-1*) election messages
  - **Number of Election messages :**

    $= N-1 + N-2 + \ldots + 1 = (N-1)*N/2 = O(N^2)$
- **Best-case**
  - Second-highest id detects leader failure
  - Sends (*N-2*) Coordinator messages
  - **Completion time:** 1 message transmission time

Impossibility?

- Since timeouts built into protocol, in asynchronous system model:
  - Protocol may never terminate => Liveness not guaranteed

- But satisfies liveness in synchronous system model where
  - **Worst-case one-way latency can be calculated = worst-case processing time + worst-case message latency**

Leader Election (Industry Systems: Google's Chubby and Apache Zookeeper)

Use of 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!

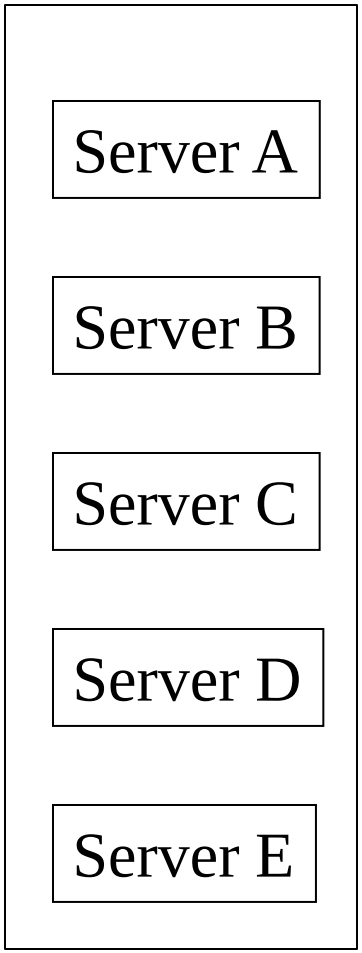
Election in Industry

- Several systems in industry use Paxos-like approaches for election
  - Paxos is a consensus protocol (safe, but eventually live)

- **Google's Chubby system**
- **Apache Zookeeper**

Election in Google Chubby

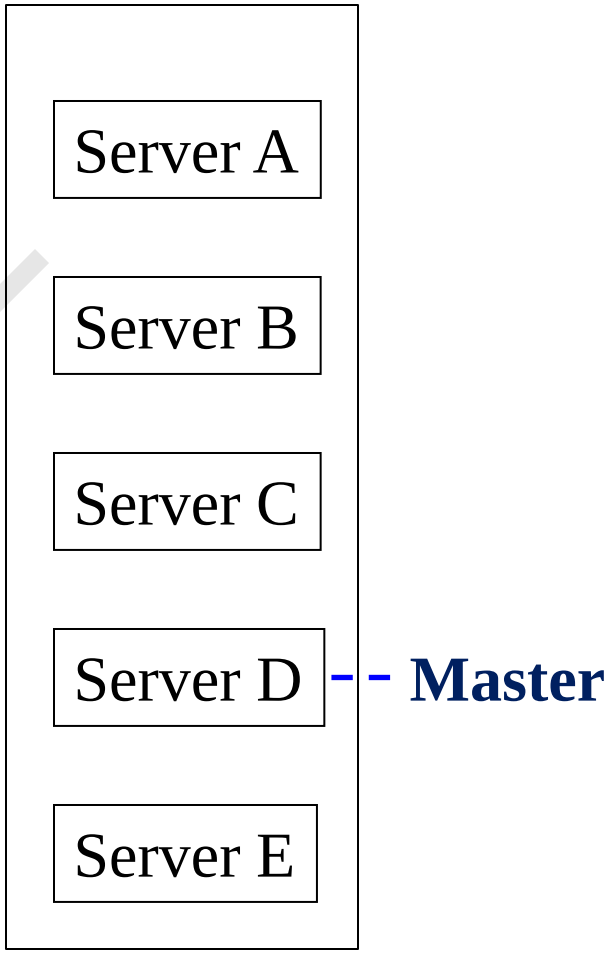
- A system for locking
- Essential part of Google's stack
  - Many of Google's internal systems rely on Chubby
  - BigTable, Megastore, etc.
- **Group of replicas**
  - Need to have a master server elected at all times



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

Election in Google Chubby (2)

- **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



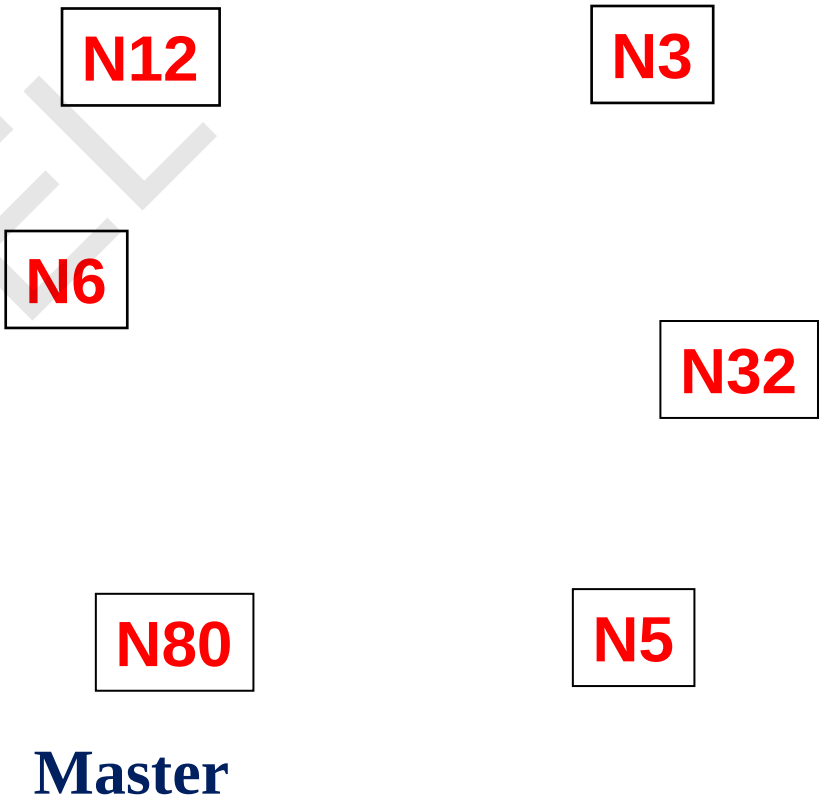
Election in Zookeeper

- Centralized service for maintaining configuration information
- **Uses a variant of Paxos called Zab (Zookeeper Atomic Broadcast)**
- Needs to keep a leader elected at all times

Reference: **http://zookeeper.apache.org/**

Election in Zookeeper (2)

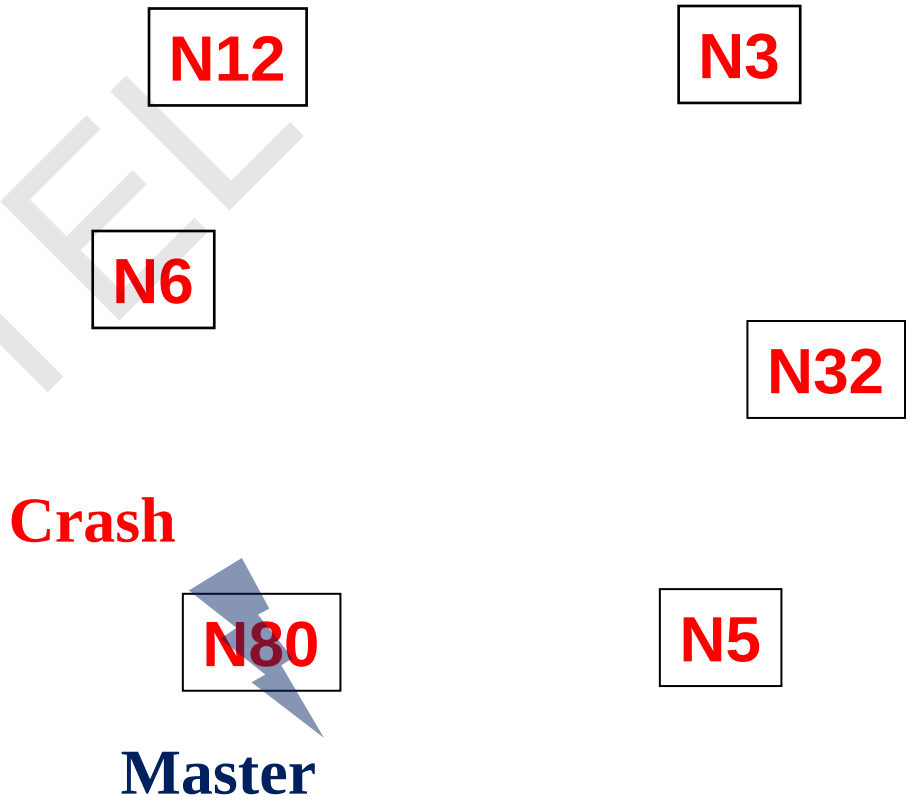
- Each server creates a new **sequence number** for itself
  - Let's say the sequence numbers are **ids**
  - Gets highest id so far (from ZK(zookeeper) file system), creates next-higher id, writes it into ZK file system
- Elect the highest-id server as leader.



Election in Zookeeper (3)

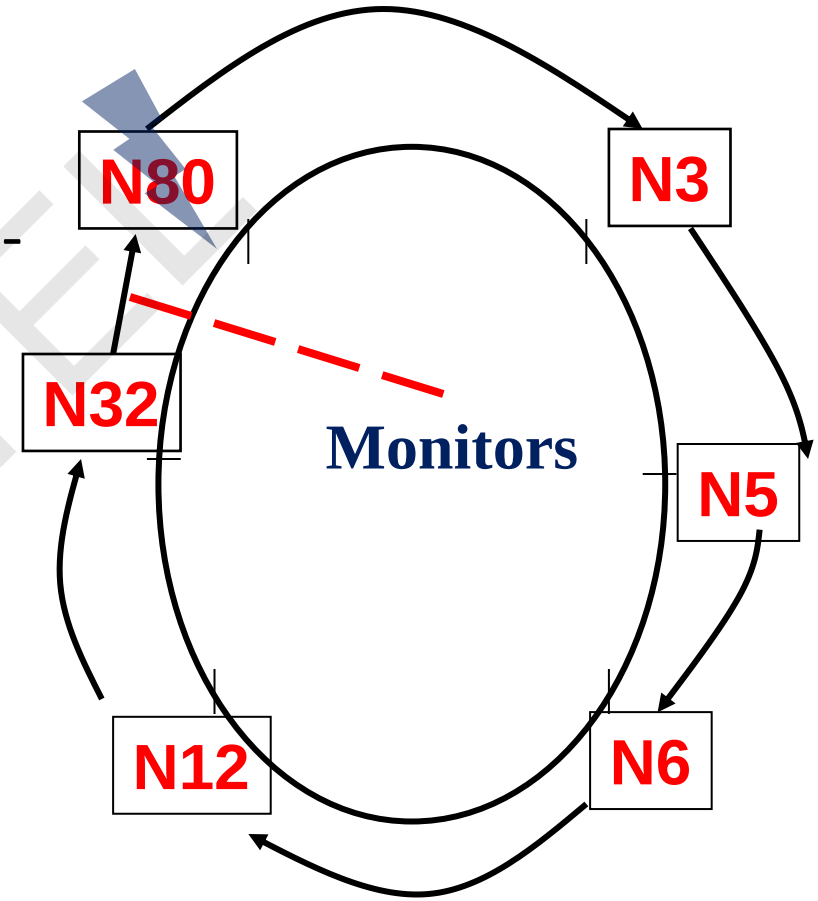
- **Failures:**

- One option: everyone monitors current master (directly or via a failure detector)
  - On failure, initiate election
  - Leads to a flood of elections
  - Too many messages



Election in Zookeeper (4)

- **Second option:** (implemented in Zookeeper)
  - Each process monitors its next-higher id process
  - **if** that successor was the leader and it has failed
    - Become the new leader
  - **else**
    - wait for a timeout, and check your successor again.



Conclusion

- Leader election an important component of many cloud computing systems

- **Classical leader election protocols**
  - Ring-based
  - Bully

- **But failure-prone**
  - Paxos-like protocols used by Google Chubby, Apache Zookeeper

Design of Zookeeper

Dr. Rajiv Misra

Associate Professor

Dept. of Computer Science & Engg.

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Content of this Lecture:

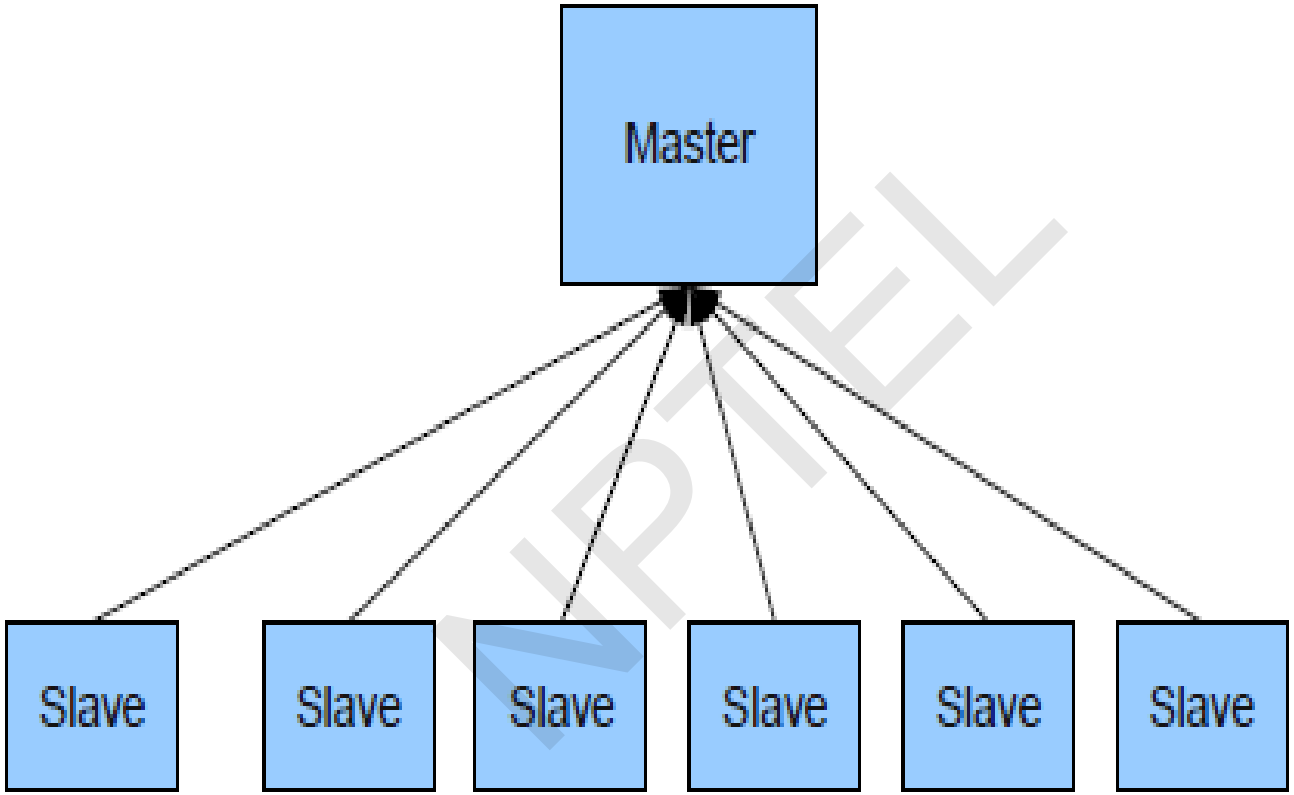
- In this lecture, we will discuss the '**design of ZooKeeper'**, which is a service for coordinating processes of distributed applications.
- We will discuss its basic fundamentals, design goals, architecture and applications.

https://zookeeper.apache.org/

ZooKeeper, why do we need it?

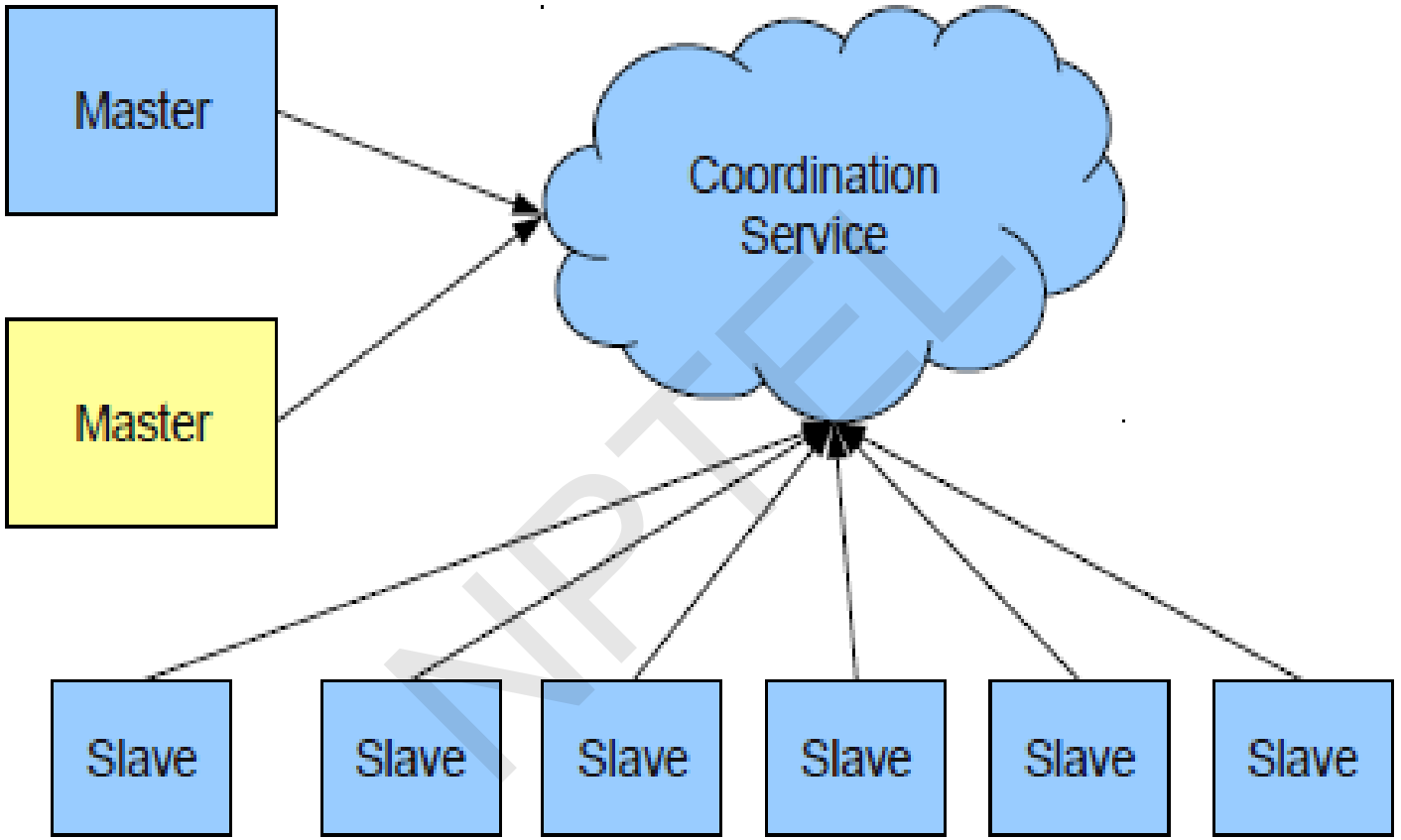
- **Coordination is important**

Classic Distributed System



- Most of the system like HDFS have one Master and couple of slave nodes and these slave nodes report to the master.

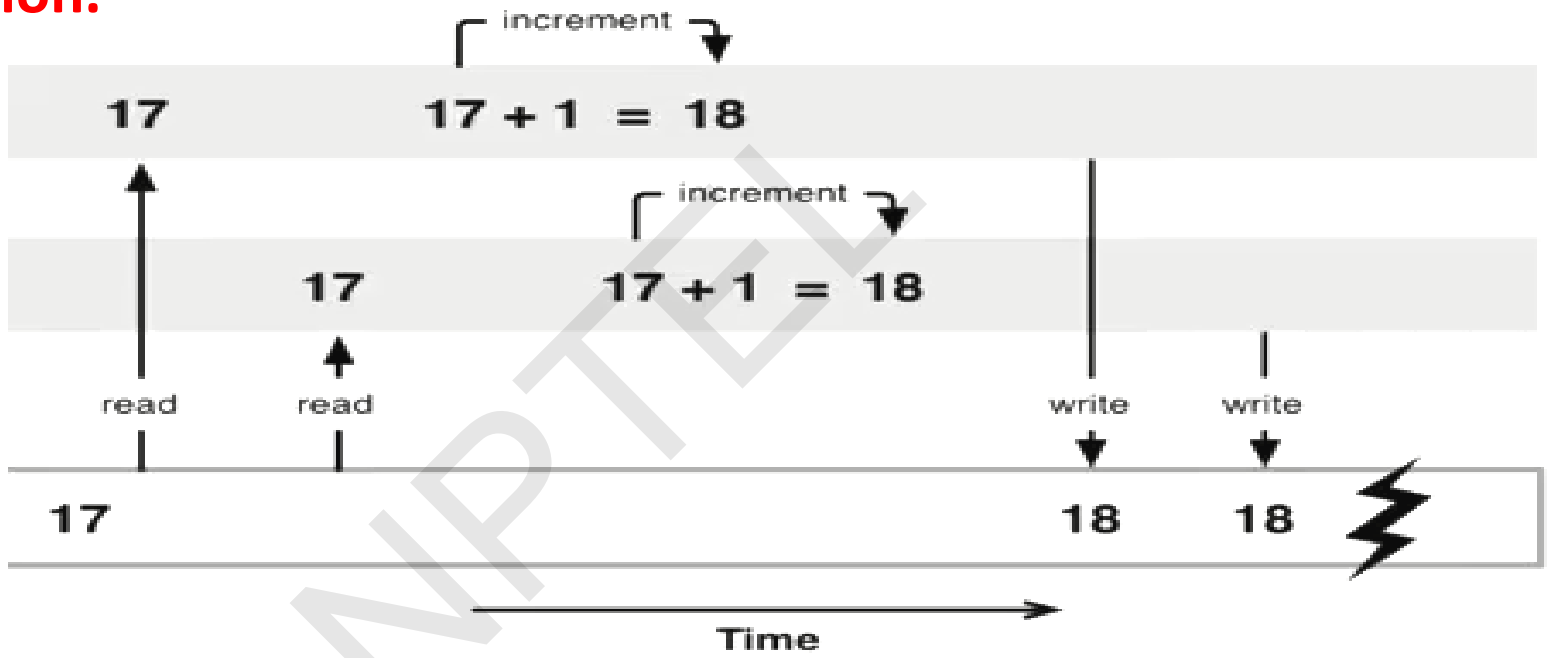
Fault Tolerant Distributed System



- Real distributed fault tolerant system have Coordination service, Master and backup master.
- If primary failed then backup works for it.

What is a Race Condition?

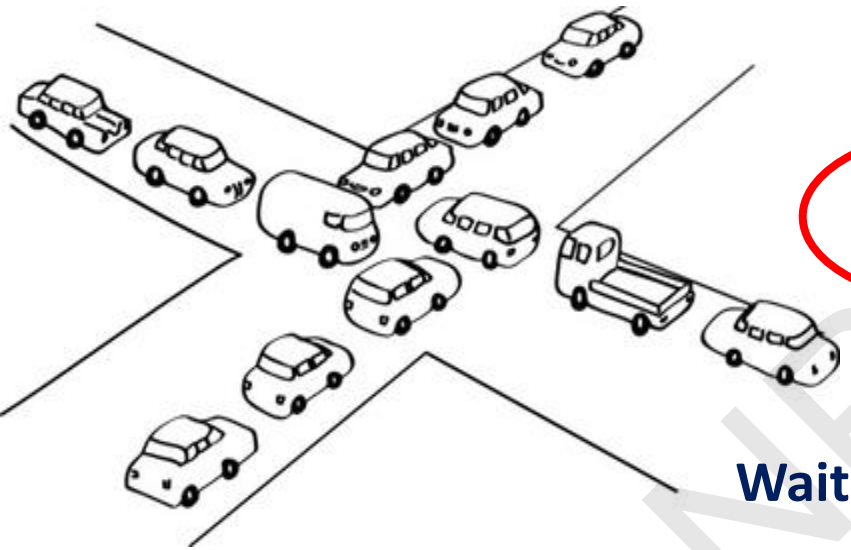
- When two processes are competing with each other causing **data corruption.**

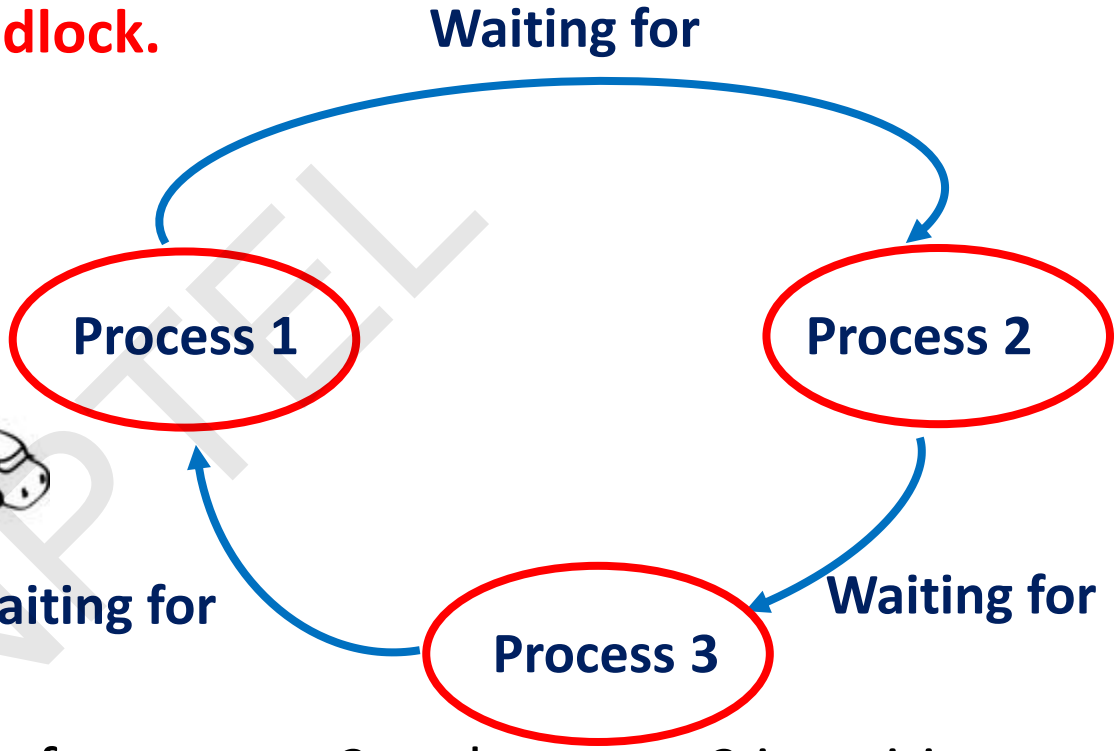


As shown in the diagram, two persons are trying to deposit 1 rs. online into the same bank account. The initial amount is 17 rs. Due to race conditions, the final amount in the bank is 18 rs. instead of 19.

What is a Deadlock?

- When two processes are waiting for each other directly or indirectly, it is called **deadlock.**

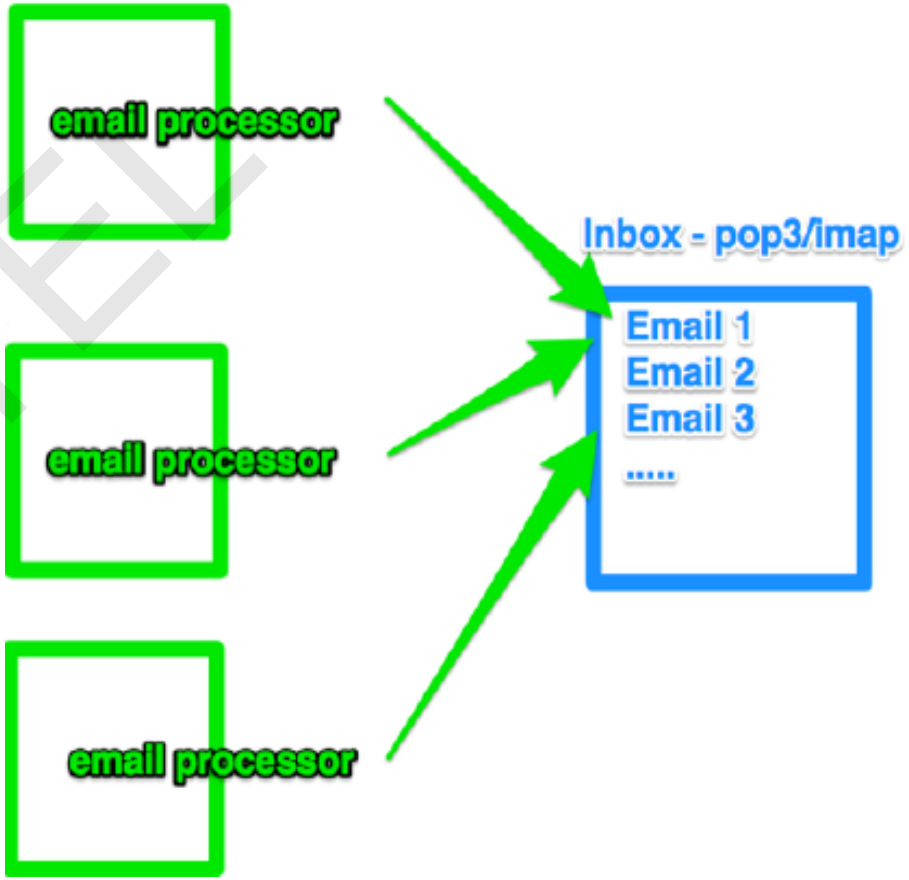




- Here, Process 1 is waiting for process 2 and process 2 is waiting for process 3 to finish and process 3 is waiting for process 1 to finish. All these three processes would keep waiting and will never end. This is called dead lock.

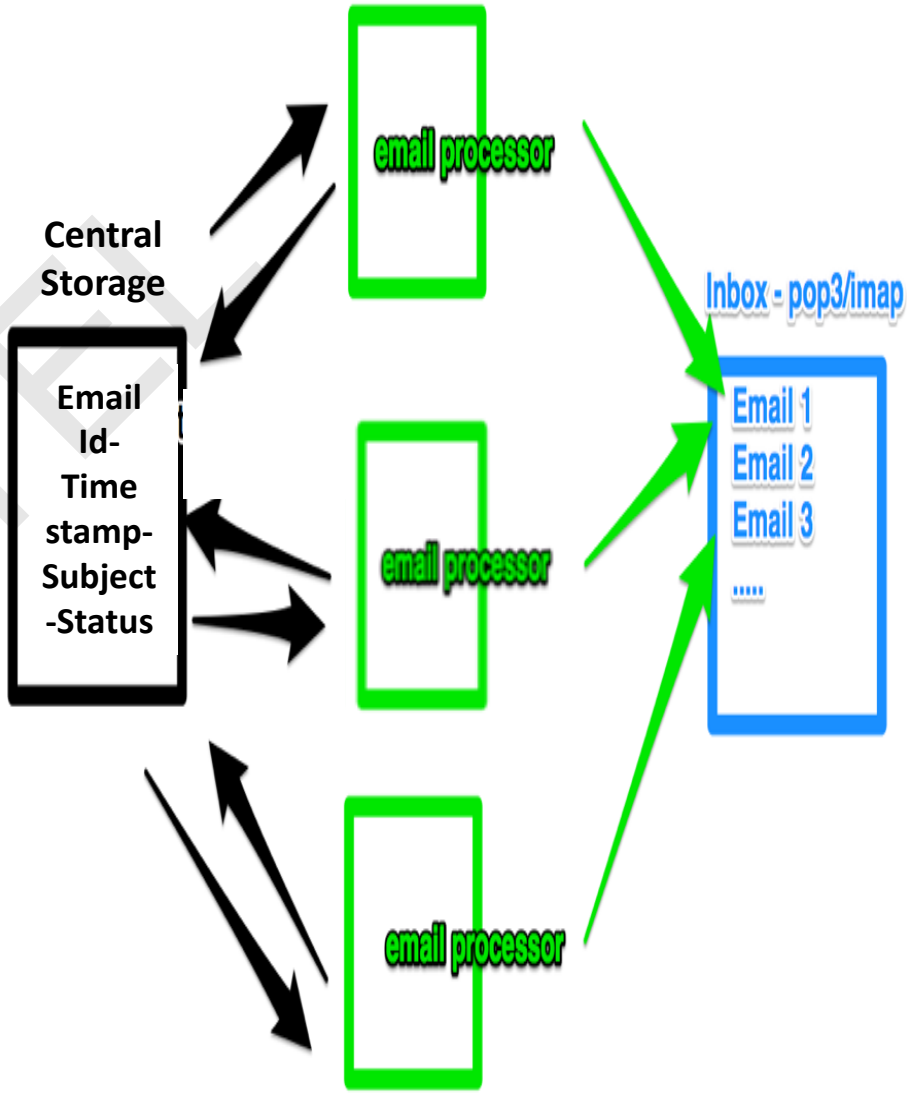
What is Coordination ?

- **How would Email Processors avoid reading same emails?**
- Suppose, there is an inbox from which we need to index emails.
- Indexing is a heavy process and might take a lot of time.
- Here, we have multiple machine which are indexing the emails. Every email has an id. You can not delete any email. You can only read an email and mark it read or unread.
- Now how would you handle the coordination between multiple indexer processes so that every email is indexed?



What is Coordination ?

- If indexers were running as multiple threads of a single process, it was easier by the way of using synchronization constructs of programming language.
- But since there are multiple processes running on multiple machines which need to coordinate, we need a central storage.
- This central storage should be safe from all concurrency related problems.
- **This central storage is exactly the role of Zookeeper.**



What is Coordination ?

- **Group membership:** Set of datanodes (tasks) belong to same group
- **Leader election:** Electing a leader between primary and backup
- **Dynamic Configuration:** Multiple services are joining, communicating and leaving (Service lookup registry)
- **Status monitoring:** Monitoring various processes and services in a cluster
- **Queuing:** One process is embedding and other is using
- **Barriers:** All the processes showing the barrier and leaving the barrier.
- **Critical sections:** Which process will go to the critical section and when?

What is ZooKeeper ?

- **ZooKeeper is a highly reliable distributed coordination kernel**, which can be used for distributed locking, configuration management, leadership election, work queues,….
- Zookeeper is a replicated service that holds the metadata of distributed applications.
- **Key attributed of such data**
  - **Small size**
  - **Performance sensitive**
  - **Dynamic**
  - **Critical**
- **In very simple words,** it is a central store of key-value using which distributed systems can coordinate. Since it needs to be able to handle the load, Zookeeper itself runs on many machines.

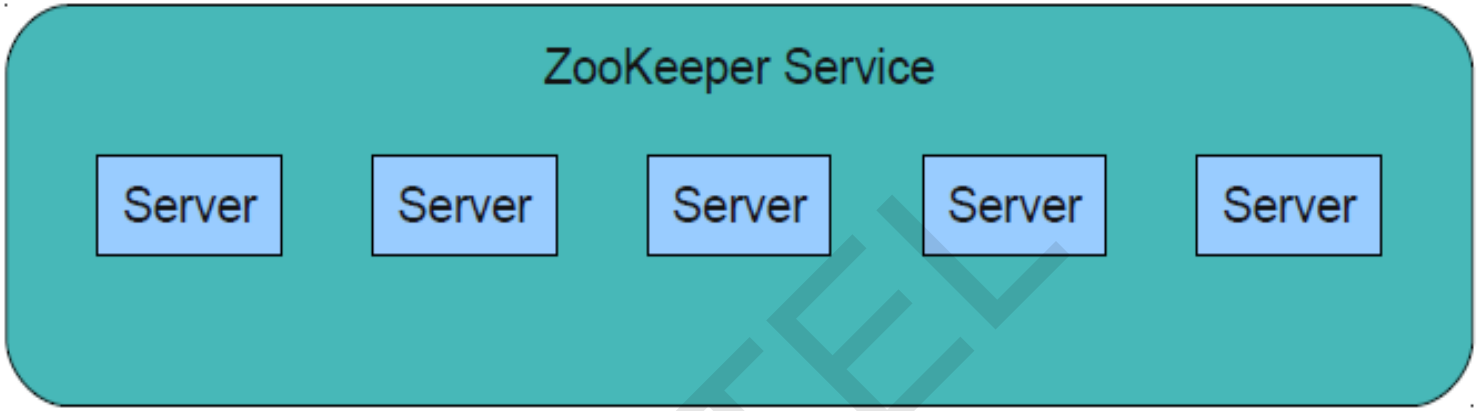
What is ZooKeeper ?

- Exposes a simple set of primitives
- Very easy to program
- Uses a data model like directory tree
- Used for
  - Synchronisation
  - Locking
  - Maintaining Configuration
- Coordination service that does not suffer from
  - Race Conditions
  - Dead Locks

Design Goals: 1. Simple

- A shared hierarchal namespace looks like standard file system
- The namespace has data nodes - znodes (similar to files/dirs)
- Data is kept in-memory
- Achieve high throughput and low latency numbers.
- **High performance**
  - Used in large, distributed systems
- **Highly available**
  - No single point of failure
- **Strictly ordered access**
  - Synchronisation

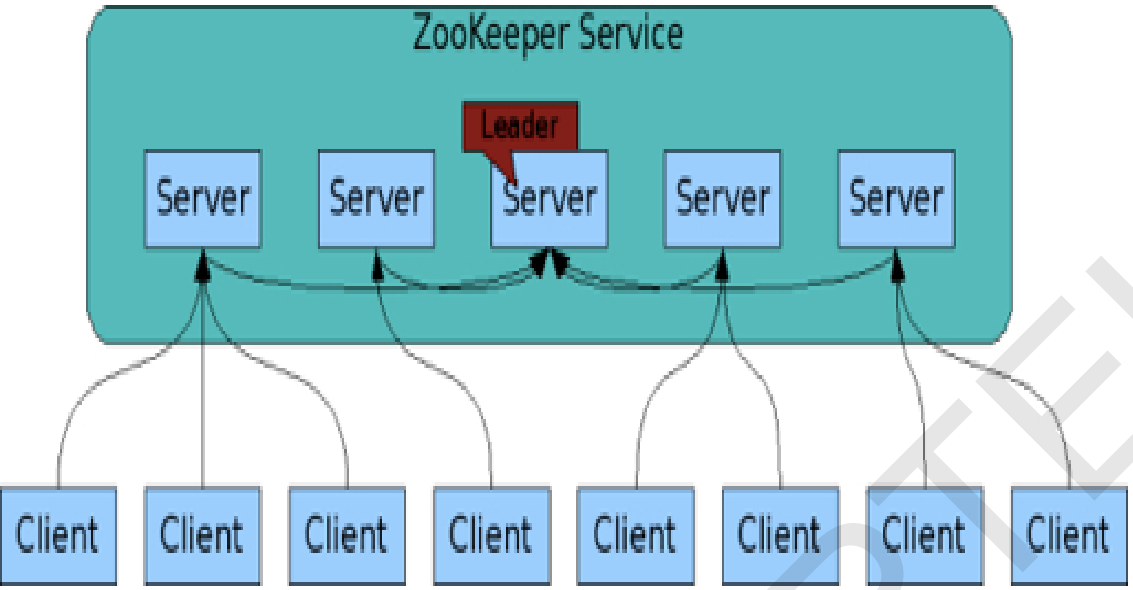
Design Goals: 2. Replicated



- All servers have a copy of the state in memory
- A leader is elected at startup
- Followers service clients, all updates go through leader
- Update responses are sent when a majority of servers have persisted the change

We need $2f+1$ machines to tolerate f failures

Design Goals: 2. Replicated



The client

- Keeps a TCP connection
- Gets watch events
- Sends heart beats.
- If connection breaks,
  - Connect to different server.

The servers

- Know each other
- Keep in-memory image of State
- Transaction Logs & Snapshots - persistent

Design Goals: 3. Ordered

- ZooKeeper stamps each update with a number

- **The number:**
  - Reflects the order of transactions.
  - used implement higher-level abstractions, such as synchronization primitives.

Design Goals: 4. Fast

Performs best where reads are more common than writes, at ratios of around 10:1.

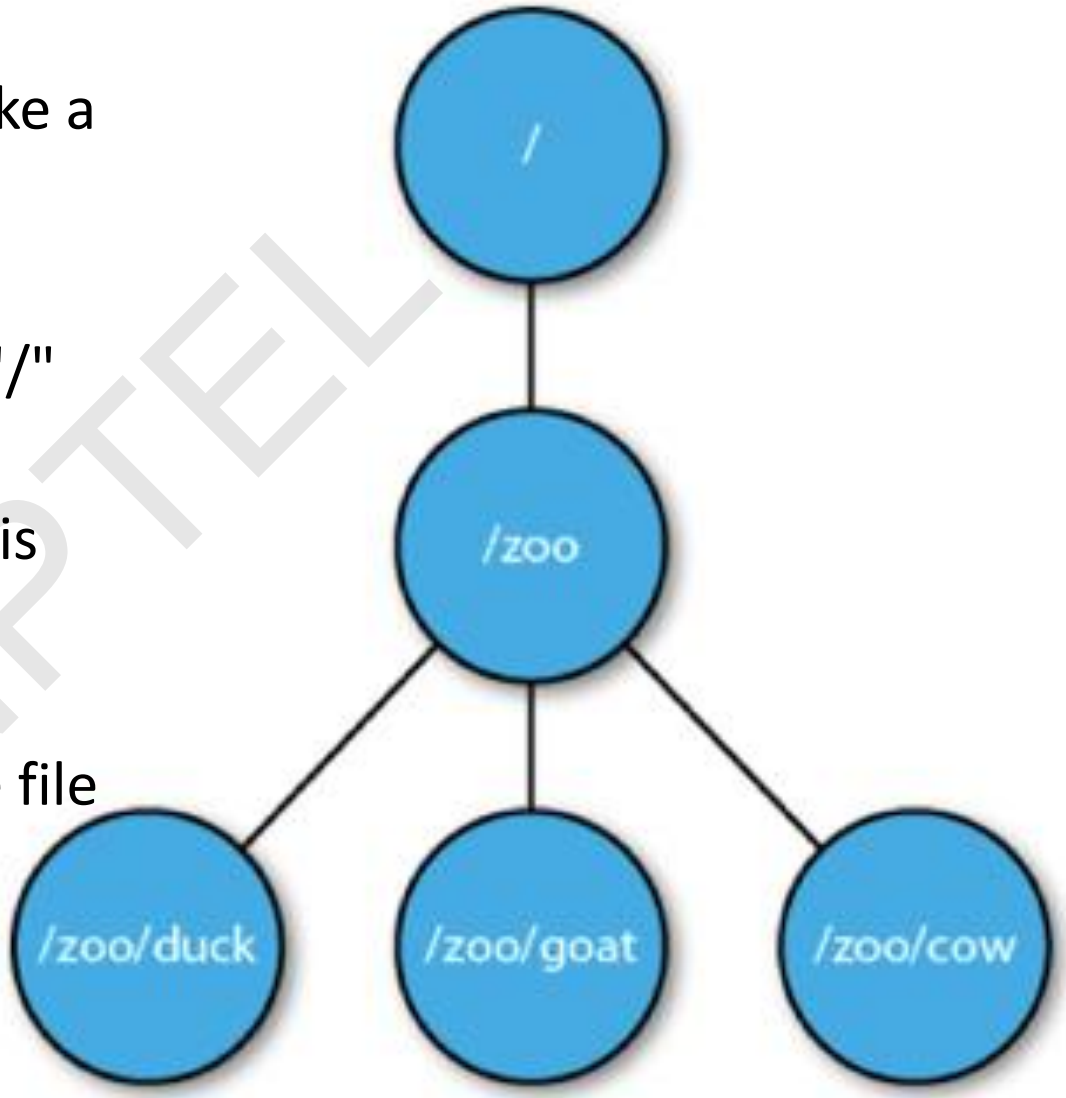
At Yahoo!, where it was created, the throughput for a ZooKeeper cluster has been benchmarked at over 10,000 operations per second for write-dominant workloads generated by hundreds of clients

Data Model

- The way you store data in any store is called **data model.**
- **Think of it as highly available fileSystem:** In case of zookeeper, think of data model as if it is a highly available file system with little differences.
- **Znode:** We store data in an entity called znode.
- **JSON data:** The data that we store should be in JSON format which Java script object notation.
- **No Append Operation:** The znode can only be updated. It does not support append operations.
- **Data access (read/write) is atomic:** The read or write is atomic operation meaning either it will be full or would throw an error if failed. There is no intermediate state like half written.
- **Znode:** Can have children

Data Model Contd…

- So, znodes inside znodes make a tree like heirarchy.
- The top level znode is "/".
- The znode "/zoo" is child of "/" which top level znode.
- duck is child znode of zoo. It is denoted as /zoo/duck
- Though "." or ".." are invalid characters as opposed to the file system.



Data Model – Znode - Types

- **Persistent**
  - Such kind of znodes remain in zookeeper untill deleted. This is the default type of znode. To create such node you can use the command: create /name_of_myznode "mydata"

- **Ephemeral**
  - Ephermal node gets deleted if the session in which the node was created has disconnected. Though it is tied to client's session but it is visible to the other users.

  - An ephermal node can not have children not even ephermal children.

Data Model – Znode - Types

- **Sequential**
  - Creates a node with a sequence number in the name
  - The number is automatically appended.

```
create -s /zoo v
Created /zoo0000000008
```

```
create -s /zoo/ v
Created /zoo/0000000003
```

```
create -s /xyz v
Created /xyz0000000009
```

```
create -s /zoo/ v
Created /zoo/0000000004
```

- The counter keeps increasing monotonically
- Each node keeps a counter

Architecture

- **Zookeeper can run in two modes: (i) Standalone and (ii) Replicated.**

(i) Standalone:

- In standalone mode, it is just running on one machine and for practical purposes we do not use stanalone mode.
- This is only for testing purposes.
- It doesn't have high availability.

(ii) Replicated:

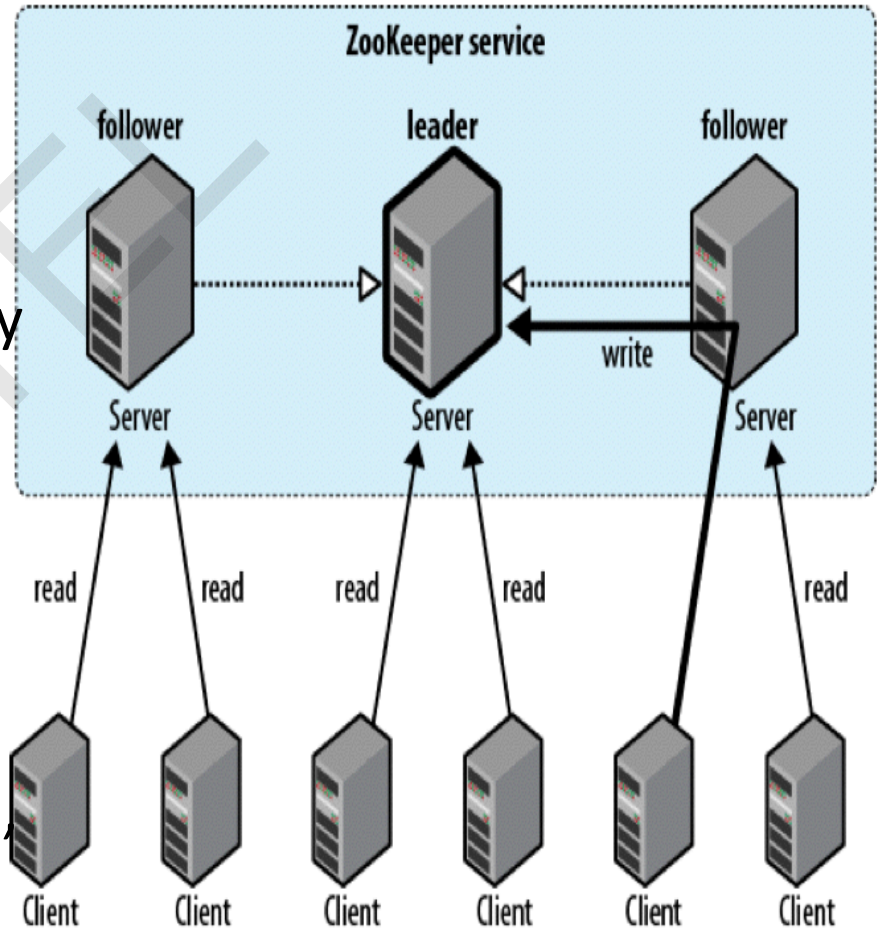
- Run on a cluster of machines called an ensemble.
- High availability
- Tolerates as long as majority.

Architecture: Phase 1

Phase 1: Leader election (Paxos Algorithm)

- The machines elect a distinguished member - leader.
- The others are termed followers.
- This phase is finished when majority sync their state with leader.
- If leader fails, the remaining machines hold election. takes 200ms.
- If the majority of the machines aren't available at any point of time, the leader automatically steps down.

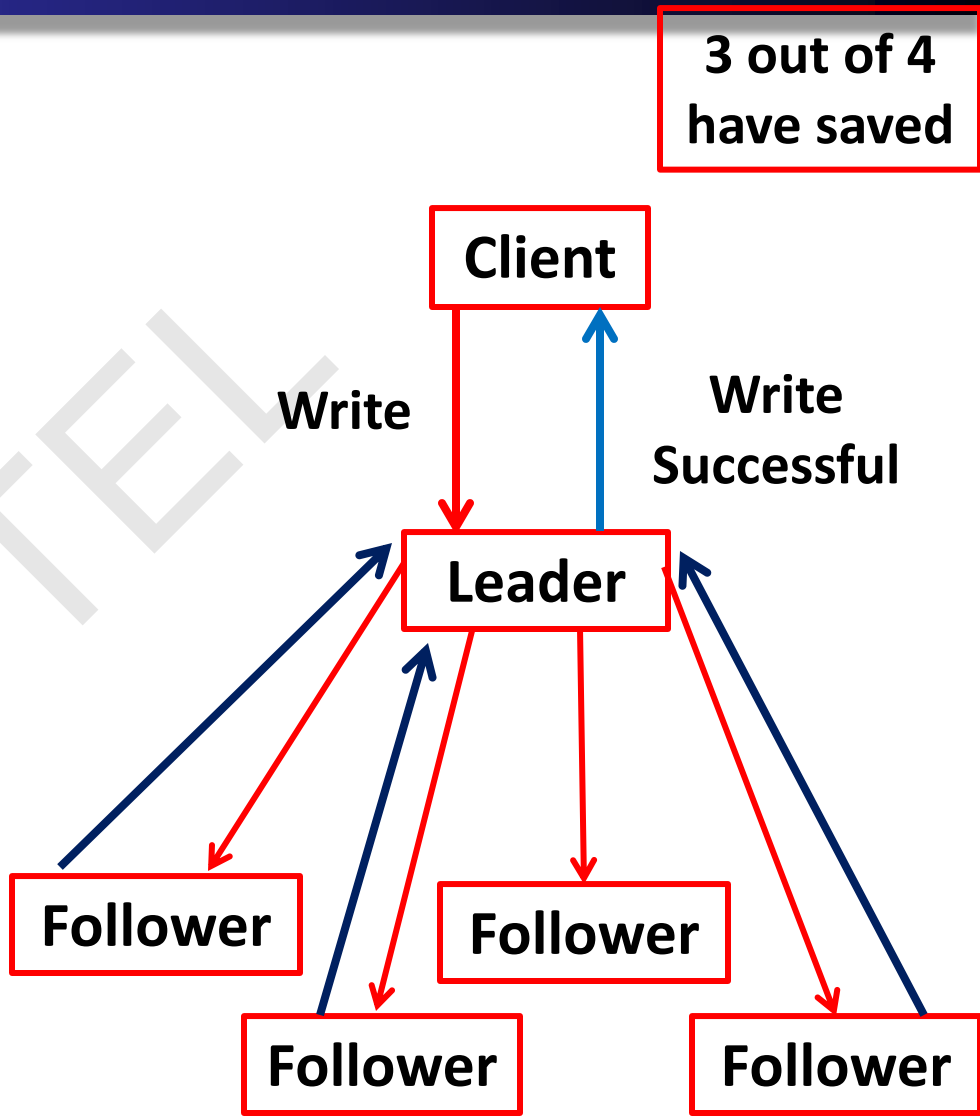
Ensemble



Architecture: Phase 2

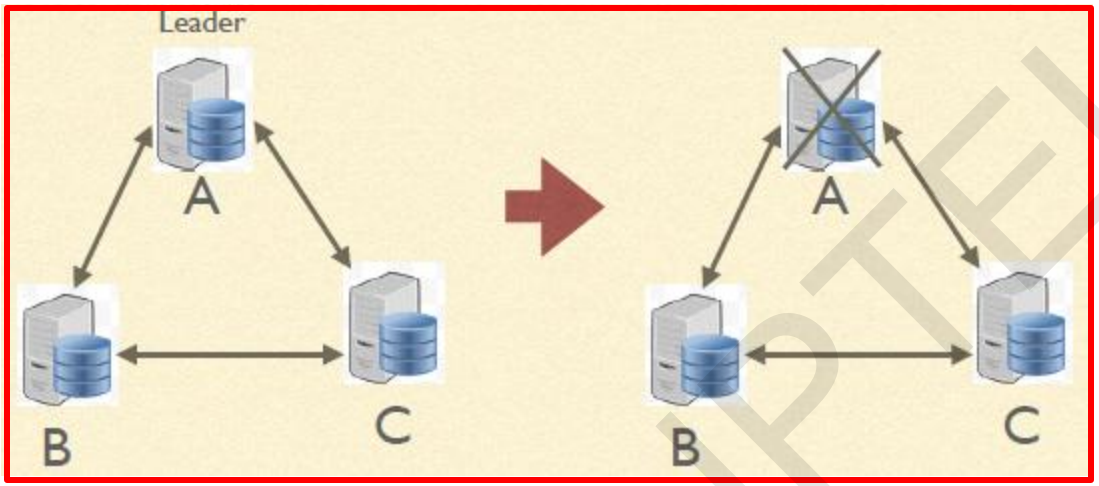
Phase 2: Atomic broadcast

- All write requests are forwarded to the leader,
- Leader broadcasts the update to the followers
- When a majority have persisted the change:
  - The leader commits the up-date
  - The client gets success response.
- The protocol for achieving consensus is atomic like two-phase commit.
- Machines write to disk before in-memory



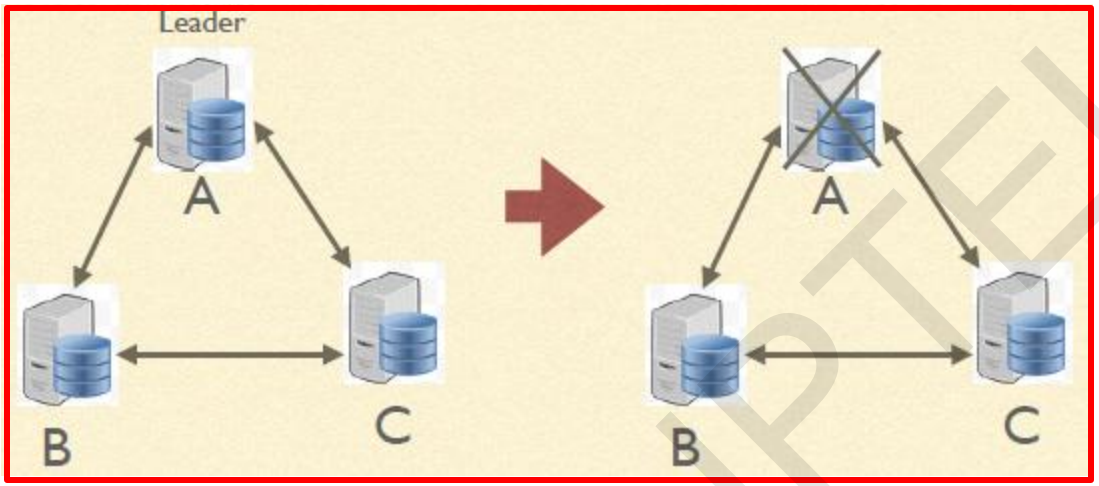
Election Demo

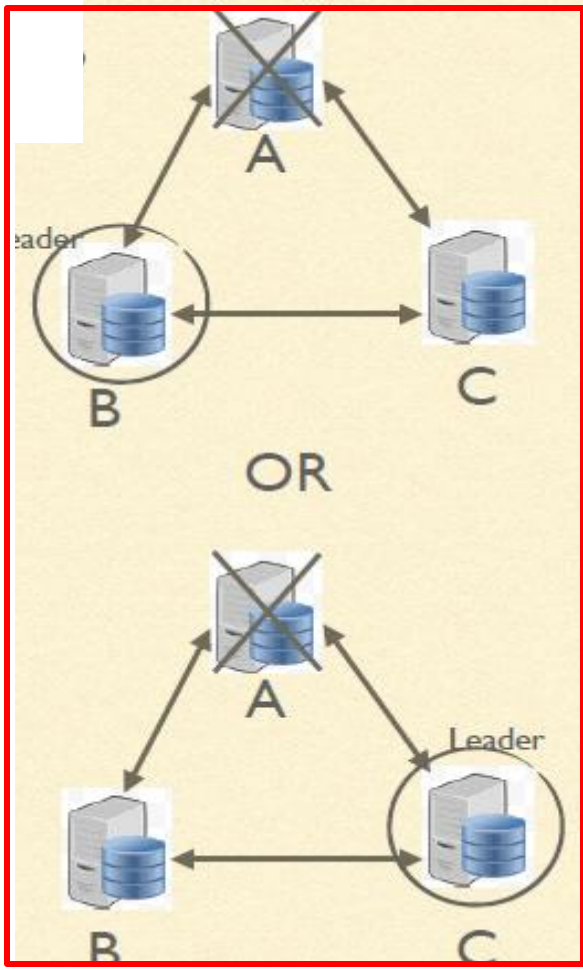
- If you have three nodes A, B, C with A as Leader. And A dies. Will someone become leader?



Election Demo

- If you have three nodes A, B, C with A as Leader. And A dies. Will someone become leader?

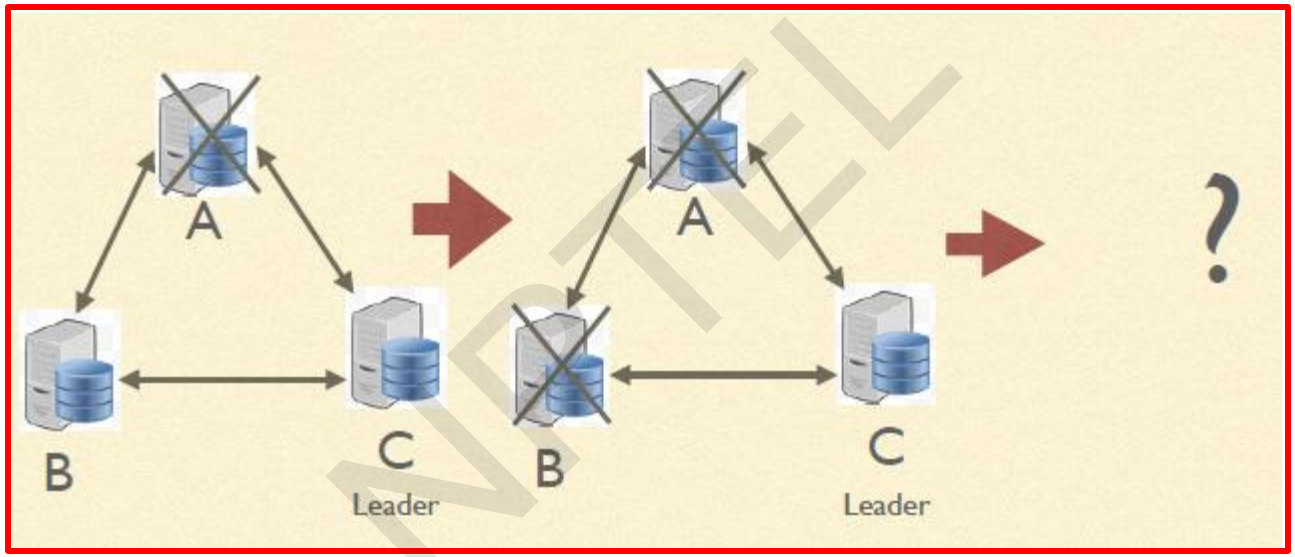




Yes. Either B or C.

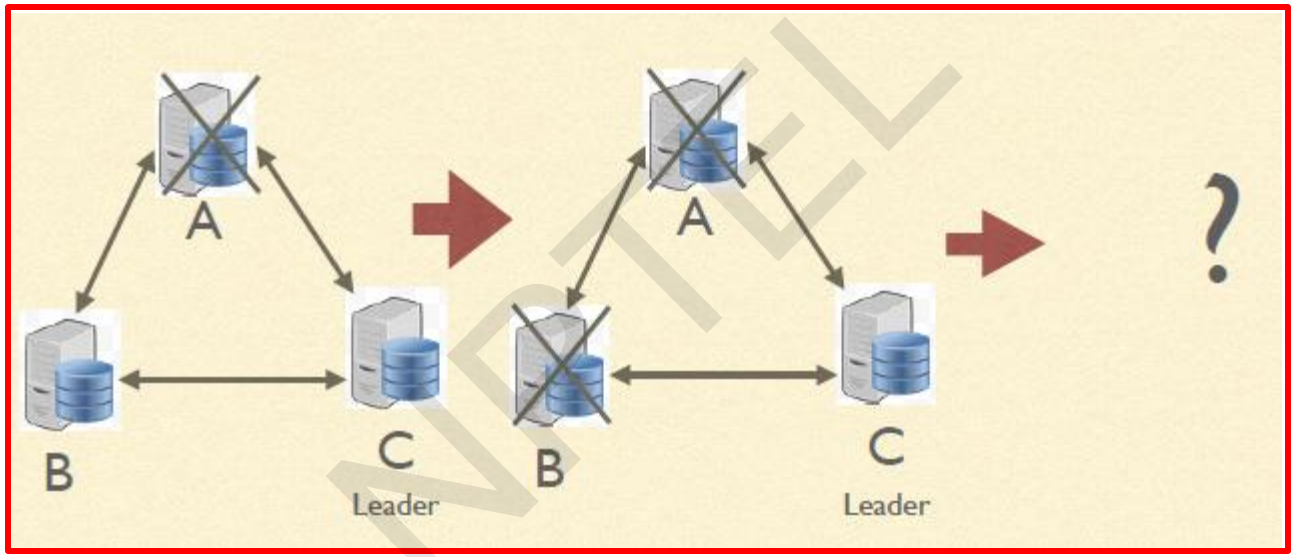
Election Demo

- If you have three nodes A, B, C And A and B die. Will C become Leader?



Election Demo

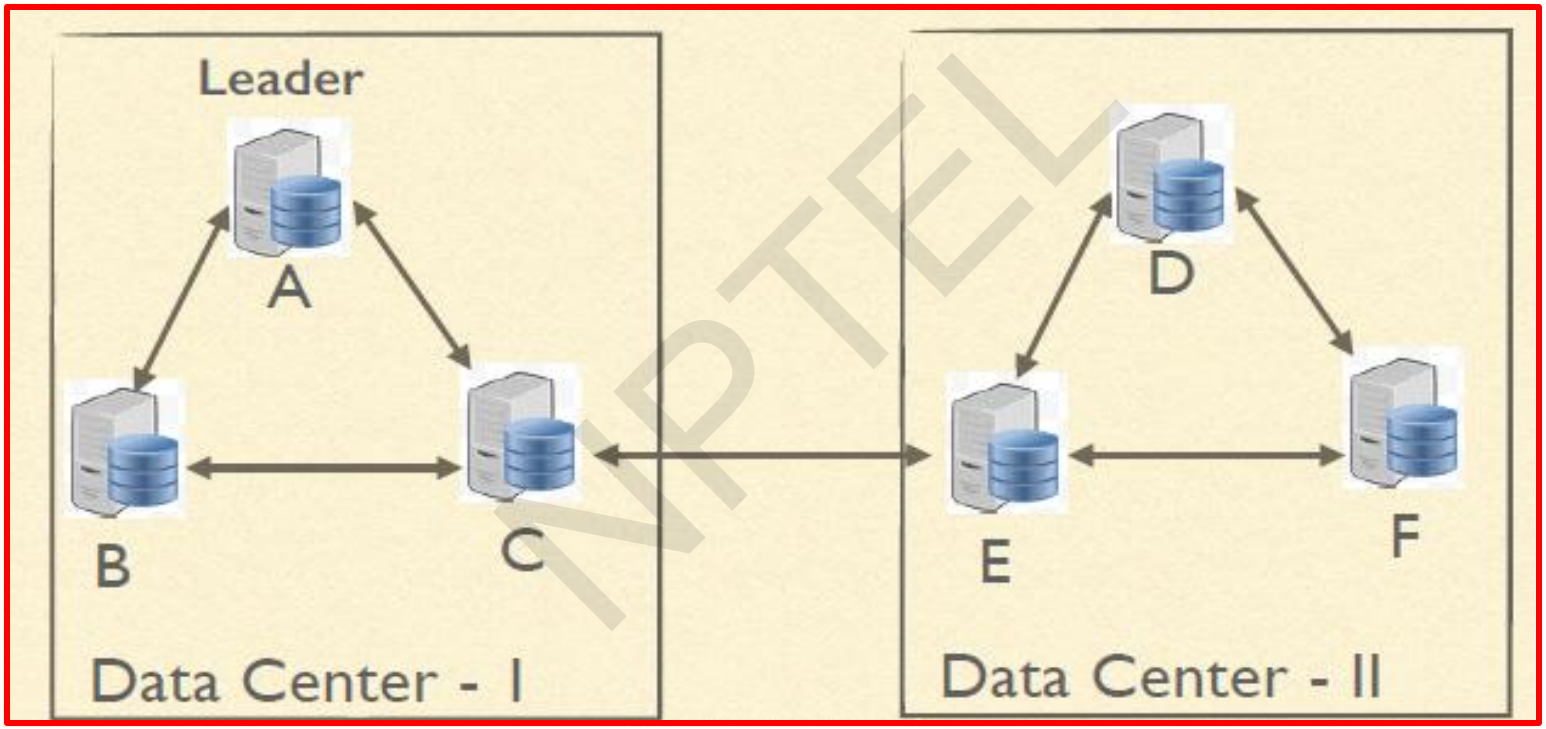
- If you have three nodes A, B, C And A and B die. Will C become Leader?



No one will become Leader.
C will become Follower.
Reason: Majority is not available.

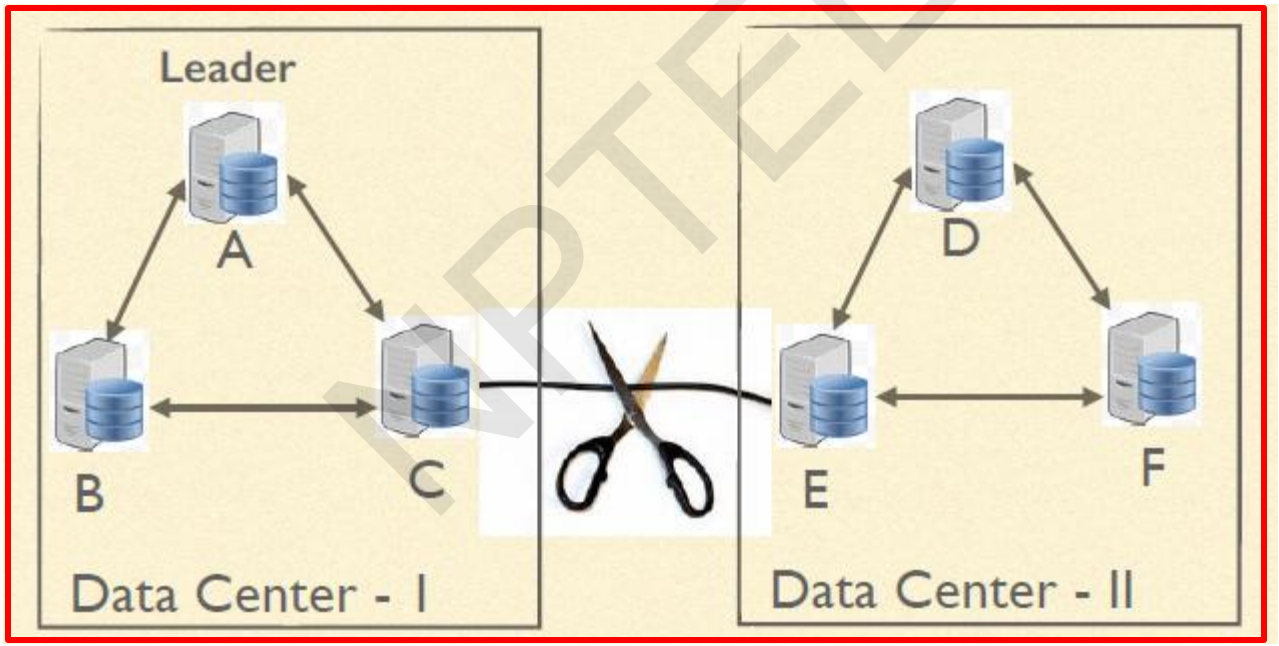
Why do we need majority?

- **Imagine:** We have an ensemble spread over two data centres.



Why do we need majority?

- **Imagine:** The network between data centres got disconnected. If we did not need majority for electing Leader,
- **What will happen?**

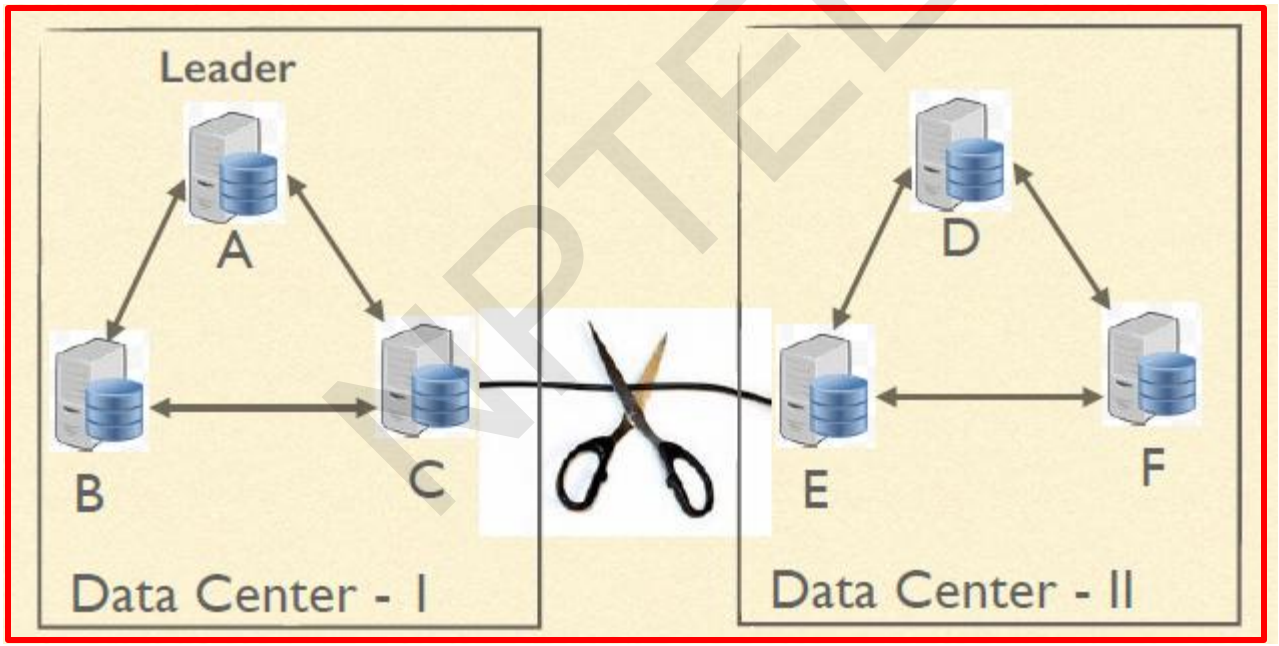


Why do we need majority?

Each data centre will have their own Leader.

No Consistency and utter Chaos.

That is why it requires majority.



Sessions

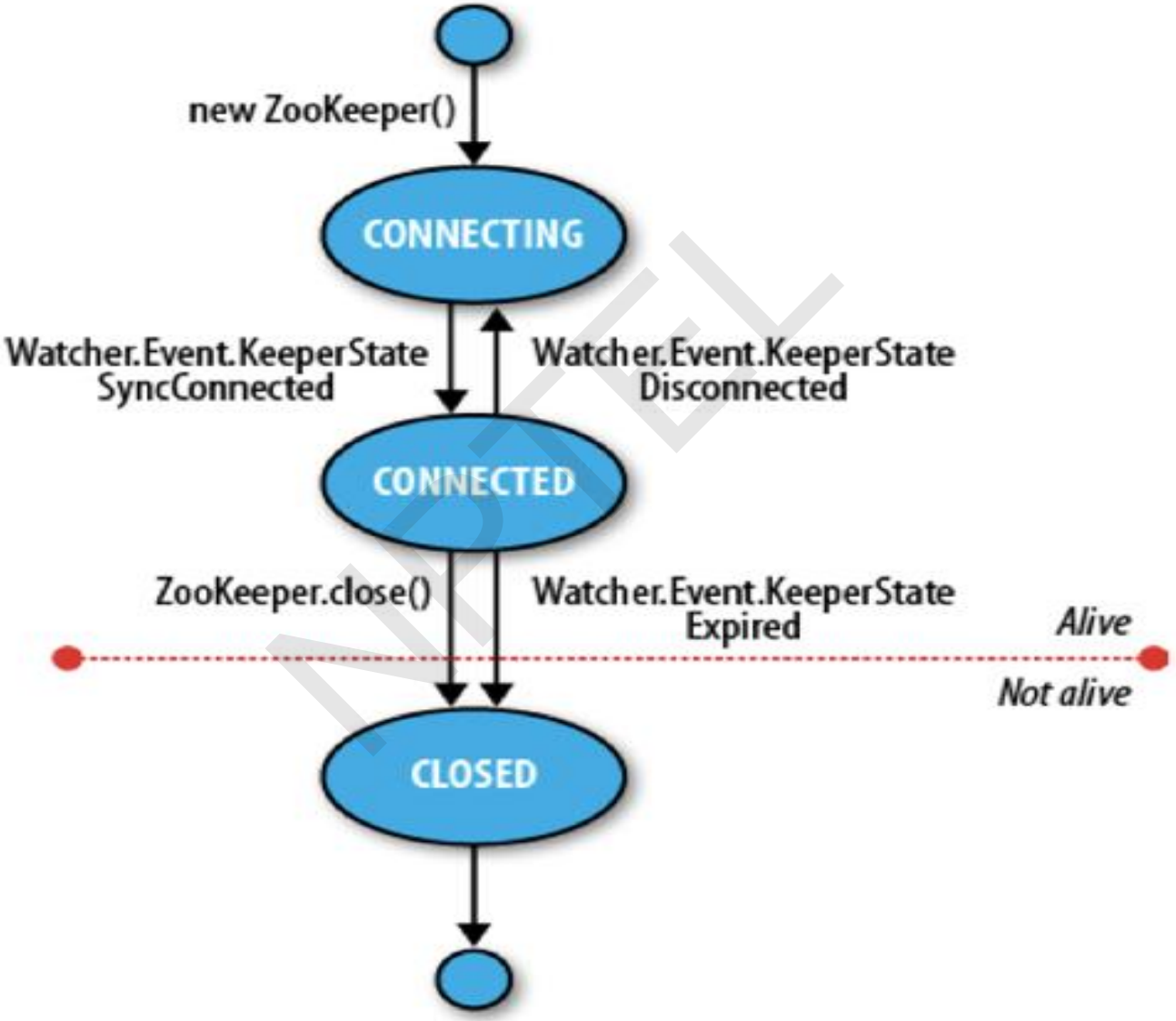
- Lets try to understand **how do the zookeeper decides to delete ephermals nodes** and takes care of session management.

- A client has list of servers in the ensemble
- It tries each until successful.
- Server creates a new session for the client.
- A session has a timeout period - decided by caller

Contd…

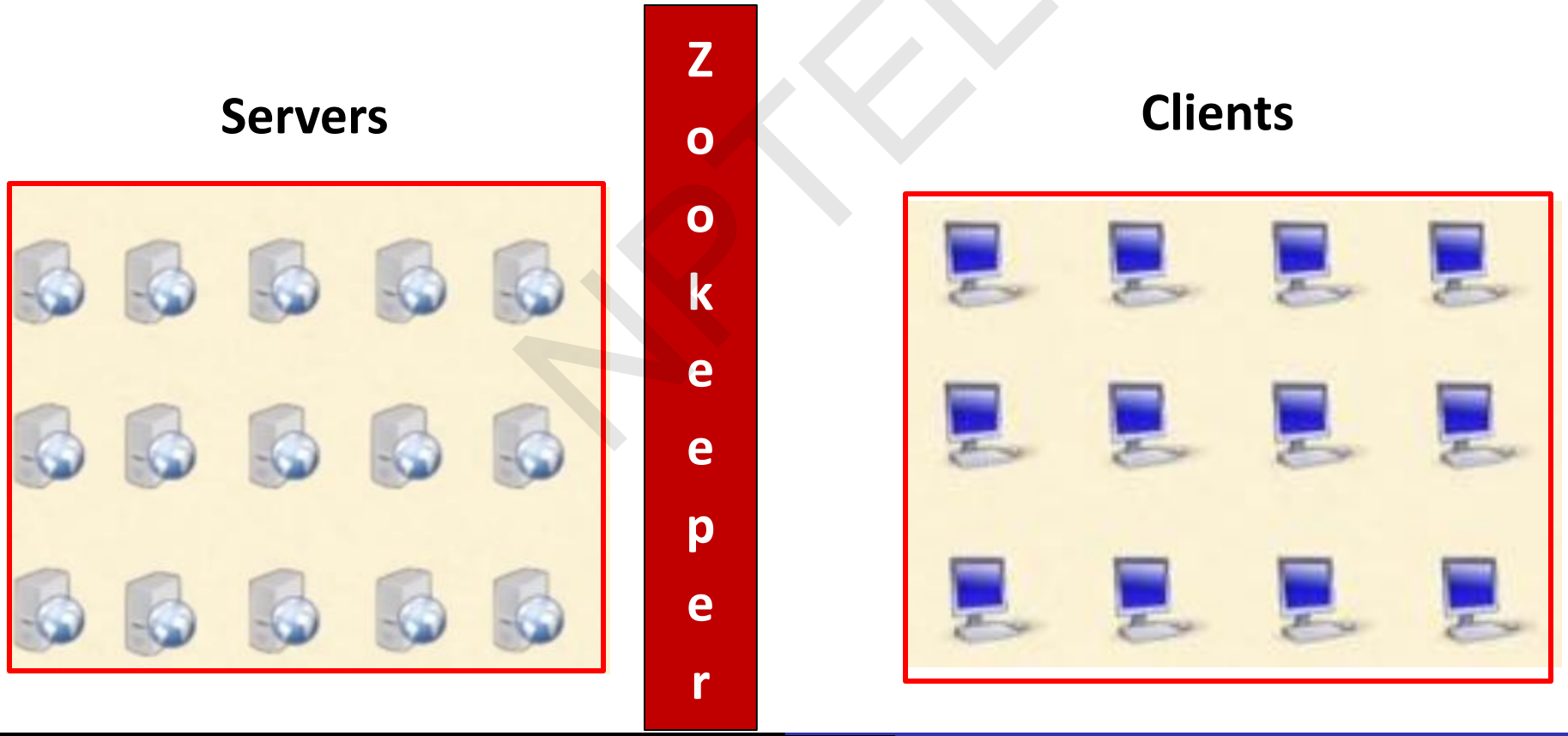
- If the server hasn’t received a request within the timeout period, it may expire the session.
- On session expire, ephermal nodes are lost
- To keep sessions alive client sends pings (heartbeats)
- Client library takes care of heartbeats
- Sessions are still valid on switching to another server
- Failover is handled automatically by the client
- Application can't remain agnostic of server reconnections - because the operations will fail during disconnection.

States



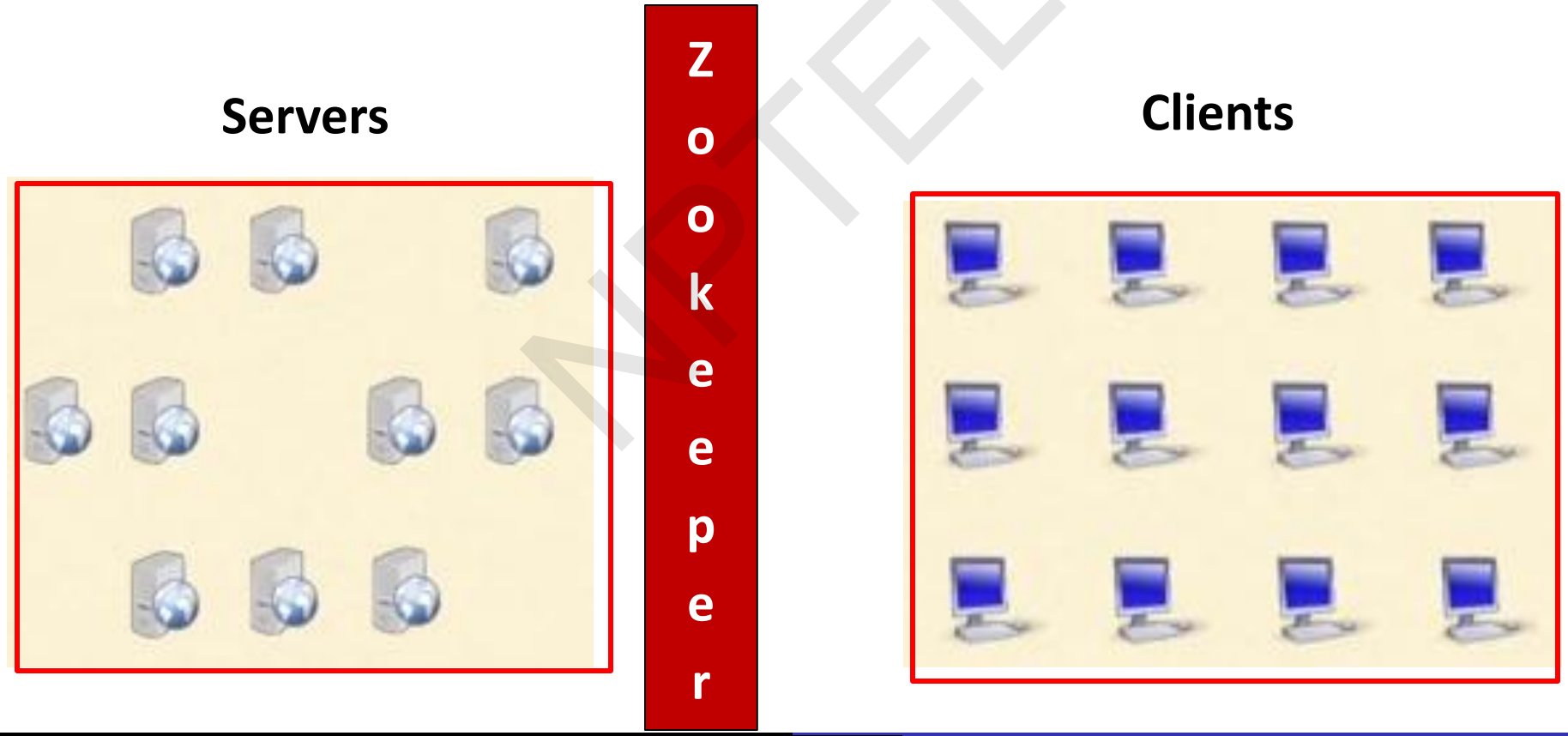
Use Case: Many Servers How do they Coordinate?

- Let us say there are many servers which can respond to your request and there are many clients which might want the service.



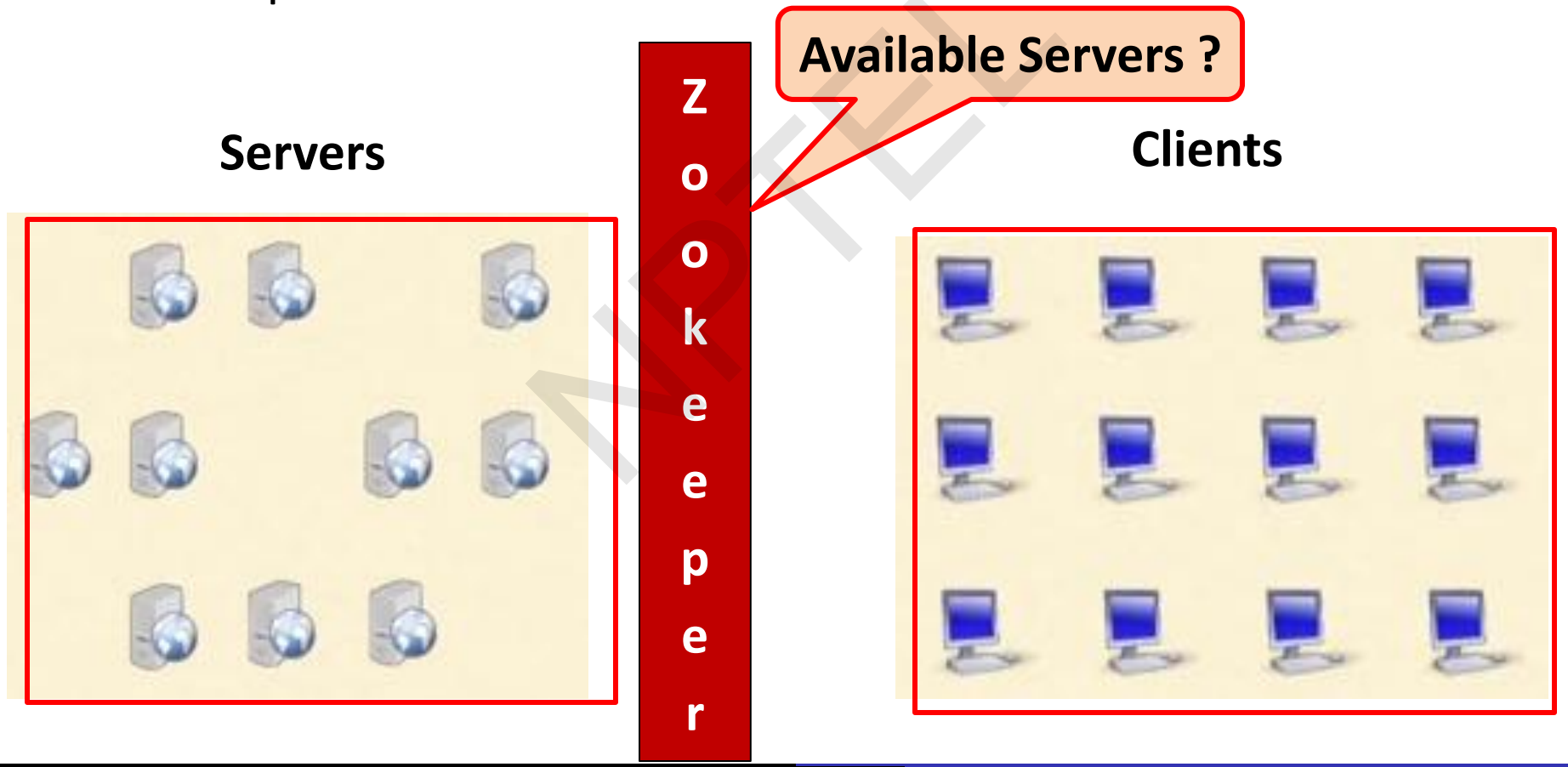
Use Case: Many Servers How do they Coordinate?

- From time to time some of the servers will keep going down. How can all of the clients can keep track of the available servers?



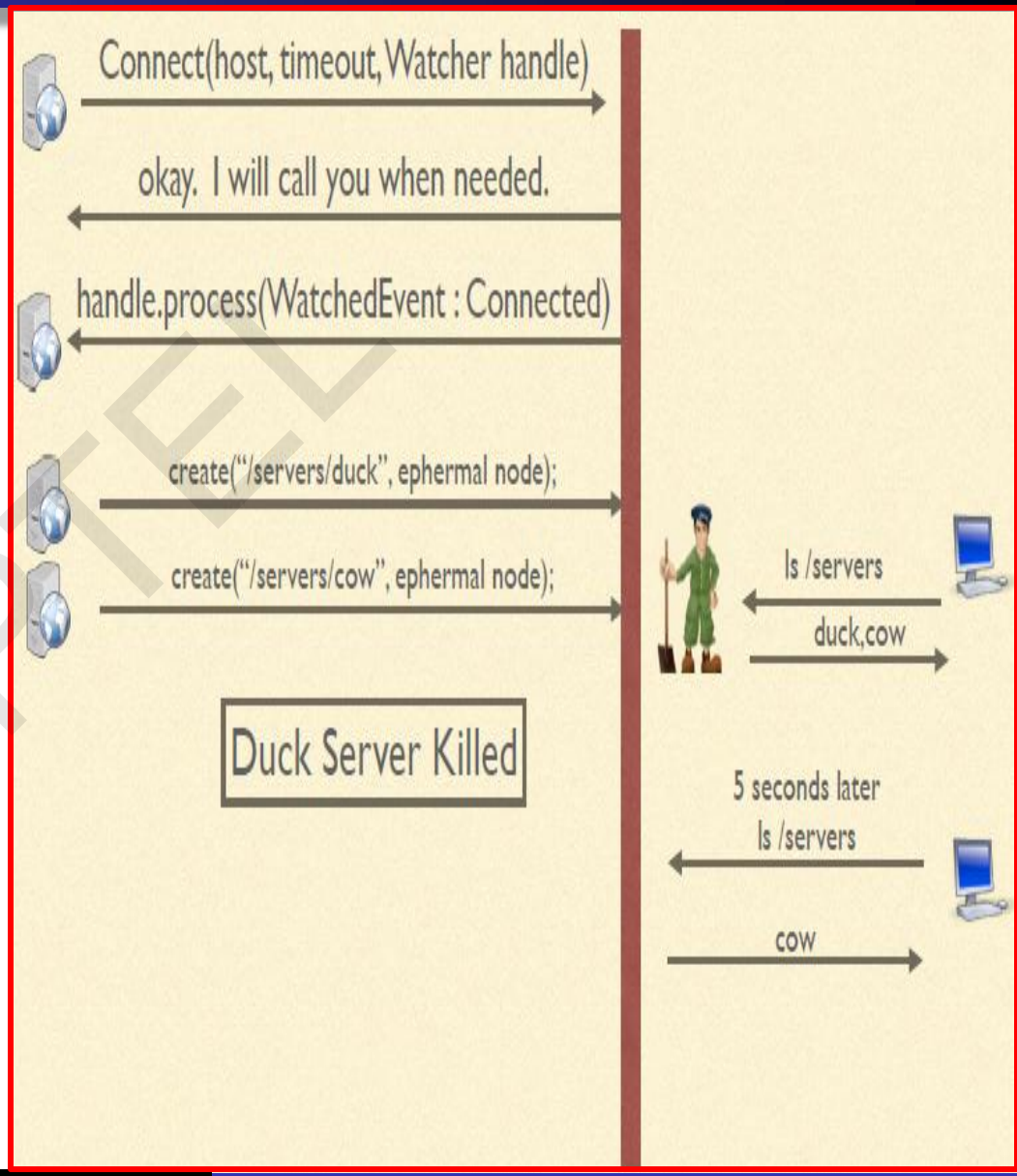
Use Case: Many Servers How do they Coordinate?

- It is very easy using zookeeper as a centeral agency. Each server will create their own ephermal znode under a particular znode say "/servers". The clients would simply query zookeeper for the most recent list of servers.



Use Case: Many Servers How do they Coordinate?

- Lets take a case of two servers and a client. The two server duck and cow created their ephermal nodes under "/servers" znode. The client would simply discover the alive servers cow and duck using command ls /servers.
- Say, a server called "duck" is down, the ephermal node will disappear from /servers znode and hence next time the client comes and queries it would only get "cow". So, the coordinations has been made heavily simplified and made efficient because of ZooKeeper.



Guarantees

- **Sequential consistency**
  - Updates from any particular client are applied in the order
- **Atomicity**
  - Updates either succeed or fail.
- **Single system image**
  - A client will see the same view of the system, The new server will not accept the connection until it has caught up.
- **Durability**
  - Once an update has succeeded, it will persist and will not be undone.
- **Timeliness**
  - Rather than allow a client to see very stale data, a server will shut down,

Operations

OPERATION	DESCRIPTION
create	Creates a znode (parent znode must exist)
delete	Deletes a znode (mustn't have children)
exists/ls	Tests whether a znode exists & gets metadata
getACL, setACL	Gets/sets the ACL for a znode getChildren/ls Gets a list of the children of a znode
getData/get, setData	Gets/sets the data associated with a znode
sync	Synchronizes a client's view of a znode with ZooKeeper

Multi Update

- Batches together multiple operations together
- Either all fail or succeed in entirety
- Possible to implement transactions
- Others never observe any inconsistent state

APIs

- Two core: Java & C
- contrib: perl, python, REST
- For each binding, sync and async available

Synch:

Public Stat exists (String path, Watcher watcher) throws KeeperException, InterruptedException

Asynch:

Public void exists (String path, Watcher watcher, StatCallback cb, Object ctx

Watches

- Clients to get notifications when a znode changes in some way
- Watchers are triggered only once
- For multiple notifications, re-register

Watch Triggers

- The read operations exists, getChildren, getData may have watches
- Watches are triggered by write ops: create, delete, setData
- **ACL (Access Control List)** operations do not participate in watches

WATCH OF …ARE TRIGGERED	WHEN ZNODE IS…
exists	created, deleted, or its data updated.
getData	deleted or has its data updated.
getChildren	deleted, or its any of the child is created or deleted

ACLs - Access Control Lists

ACL Determines who can perform certain operations on it.

- **ACL is the combination**
  - authentication scheme,
  - an identity for that scheme,
  - and a set of permissions

- **Authentication Scheme**
  - **digest -** The client is authenticated by a username & password.
  - **sasl -** The client is authenticated using Kerberos.
  - **ip -** The client is authenticated by its IP address.

Use Cases

Building a reliable configuration service

- A Distributed lock service

  Only single process may hold the lock

When Not to Use?

1. To store big data because:
   - The number of copies == number of nodes
   - All data is loaded in RAM too
   - Network load of transferring all data to all Nodes

2. Extremely strong consistency

ZooKeeper Applications: The Fetching Service

- **The Fetching Service:** Crawling is an important part of a search engine, and Yahoo! crawls billions of Web documents. The Fetching Service (FS) is part of the Yahoo! crawler and it is currently in production. Essentially, it has master processes that command page-fetching processes.

- The master provides the fetchers with configuration, and the fetchers write back informing of their status and health. The main advantages of using ZooKeeper for FS are recovering from failures of masters, guaranteeing availability despite failures, and decoupling the clients from the servers, allowing them to direct their request to healthy servers by just reading their status from ZooKeeper.

- Thus, FS uses ZooKeeper mainly to manage configuration metadata, although it also uses Zoo- Keeper to elect masters (leader election).

ZooKeeper Applications: Katta

- **Katta:** It is a distributed indexer that uses Zoo- Keeper for coordination, and it is an example of a non- Yahoo! application. Katta divides the work of indexing using shards.
- A master server assigns shards to slaves and tracks progress. Slaves can fail, so the master must redistribute load as slaves come and go.
- The master can also fail, so other servers must be ready to take over in case of failure. Katta uses ZooKeeper to track the status of slave servers and the master (**group membership**), and to handle master failover (**leader election**).
- Katta also uses ZooKeeper to track and propagate the assignments of shards to slaves (**configuration management**).

ZooKeeper Applications: Yahoo! Message Broker

- **Yahoo! Message Broker: (YMB)** is a distributed publish-subscribe system. The system manages thousands of topics that clients can publish messages to and receive messages from. The topics are distributed among a set of servers to provide scalability.
- Each topic is replicated using a primary-backup scheme that ensures messages are replicated to two machines to ensure reliable message delivery. The servers that makeup YMB use a shared-nothing distributed architecture which makes coordination essential for correct operation.
- YMB uses ZooKeeper to manage the distribution of topics (configuration metadata), deal with failures of machines in the system (failure detection and group membership), and control system operation.

ZooKeeper Applications: Yahoo! Message Broker

- Figure, shows part of the znode data layout for YMB.
- Each broker domain has a znode called nodes that has an ephemeral znode for each of the active servers that compose the YMB service.
- Each YMB server creates an ephemeral znode under nodes with load and status information providing both group membership and status information through ZooKeeper. of YMB.

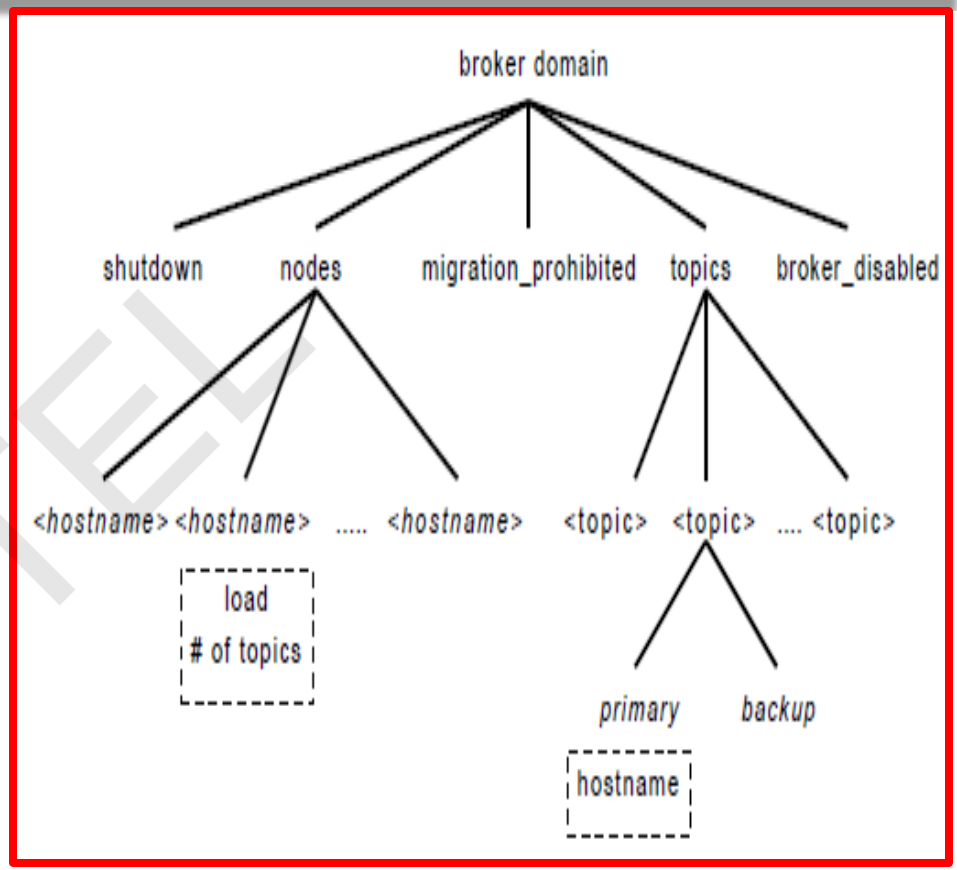


Figure: The layout of Yahoo! Message Broker (YMB) structures in ZooKeeper

ZooKeeper Applications: Yahoo! Message Broker

- The topics directory has a child znode for each topic managed by YMB.
- These topic znodes have child znodes that indicate the primary and backup server for each topic along with the subscribers of that topic.
- The primary and backup server znodes not only allow servers to discover the servers in charge of a topic, but they also manage leader election and server crashes.

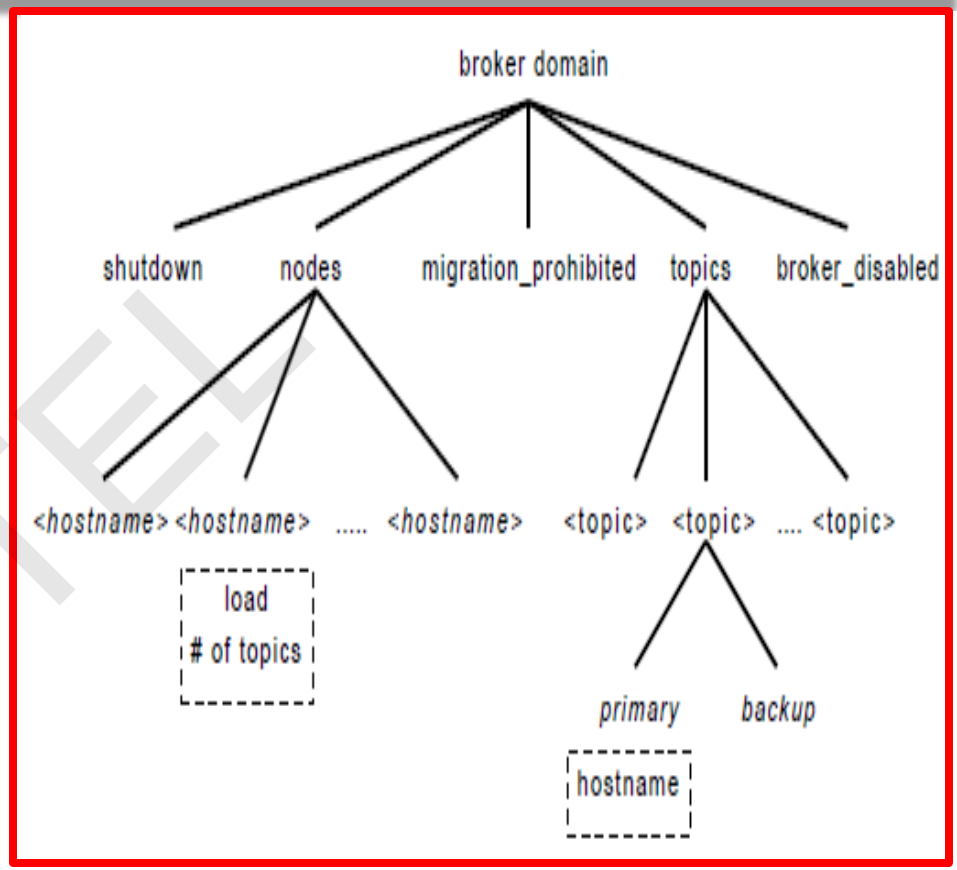


Figure: The layout of Yahoo! Message Broker (YMB) structures in ZooKeeper

More Details

ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{phunt,mahadev}@yahoo-inc.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{fpj,breed}@yahoo-inc.com

See: https://zookeeper.apache.org/

Conclusion

- **ZooKeeper** takes a wait-free approach to the problem of coordinating processes in distributed systems, by exposing wait-free objects to clients.

- **ZooKeeper** achieves throughput values of hundreds of thousands of operations per second for read-dominant workloads by using fast reads with watches, both of which served by local replicas.

- In this lecture, the basic fundamentals, design goals, architecture and applications of ZooKeeper.