CLOUD COMPUTING
Synchronization & Coordination

Zeinab Zali
Isfahan University Of Technology

References: -Cloud computing: Theory and practice, Chapter 3
-Distributed systems: concepts and design, George Coulouris, Chapter 15
-https://www.cs.rutgers.edu/~pxk/417/notes/paxos.html
-

Problem statement

- Goal:
  - for a set of processes to coordinate their actions or to agree on one or more value
  - The computers must be able to do so even where there is no fixed master-slave relationship between the components
- Example:
  - Synchronizing hadoop cluster nodes for doing a single task
  - leader/master election
  - Use barriers to block processing of a set of nodes until a condition is met

Failure assumptions and failure detectors

- Each pair of processes is connected by reliable channels
  - although the underlying network components may suffer failures, the processes use a reliable communication protocol that masks these failures – for example, by re-transmitting missing or corrupted messages
- No process failure implies a threat to the other processes’ ability to communicate.
  - This means that none of the processes depends upon another to forward messages.
- Processes may fail only by crashing

Distributed Mutual
Exclusion

Critical section problem

- If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources
- In a distributed system, however, neither shared variables nor facilities supplied by a single local kernel can be used to solve it, in general.
- We require a solution to distributed mutual exclusion: one that is based solely on message passing

executing a critical section

- The application-level protocol for executing a critical section is as follows:

```
enter()               // enter critical section – block if necessary
ResourceAccesses()    // access shared resources in critical section
exit()                // leave critical section – other processes
                      // may now enter
```

Essential requirements for mutual exclusion

- ME1 (safety): At most one process may execute in the critical section (CS) at a time.
- ME2 (liveness): Requests to enter and exit the critical section eventually succeed
  - **Deadlock:** involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section
  - **Starvation:** the indefinite postponement of entry for a process that has requested it (→ no fairness)
- ME3: (→ ordering): If one request to enter the CS happened-before another, then entry to the CS is granted in that order (no ME3 → no fairness)

Evaluating the performance of ME algorithms

- The bandwidth consumed: it is proportional to the number of messages sent in each entry and exit operation;
- Delay: the client delay incurred by a process at each entry and exit operation;
- Algorithm's throughput: This is the rate at which the collection of processes as a whole can access the critical section
  - We measure the effect using the synchronization delay between one process exiting the critical section and the next process entering it; the throughput is greater when the synchronization delay is shorter.

Multicast synchronization

- Basic idea: processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message

Maekawa's voting algorithm

- A 'candidate' process must collect sufficient votes to enter (but not all the votes like previous multicast method)

Maekawa associated a *voting set* V_i with each process p_i $(i = 1, 2, \ldots, N)$, where $V_i \subseteq \{p_1, p_1, \ldots, p_N\}$. The sets V_i are chosen so that, for all $i, j = 1, 2, \ldots, N$:

- $p_i \in V_i$
- $V_i \cap V_j \neq \emptyset$ – there is at least one common member of any two voting sets
- $|V_i| = K$ – to be fair, each process has a voting set of the same size
- Each process p_j is contained in M of the voting sets V_i.

Figure 15.6
Maekawa's algorithm

On initialization

state := RELEASED;
voted := FALSE;

For p_i to enter the critical section

state := WANTED;
Multicast *request* to all processes in V_i;
Wait until (number of replies received = K);
state := HELD;

On receipt of a request from p_i at p_j

if (*state* = HELD *or voted* = TRUE)
then
queue *request* from p_i without replying;
else
send *reply* to p_i;
voted := TRUE;
end if

For p_i to exit the critical section

state := RELEASED;
Multicast *release* to all processes in V_i;

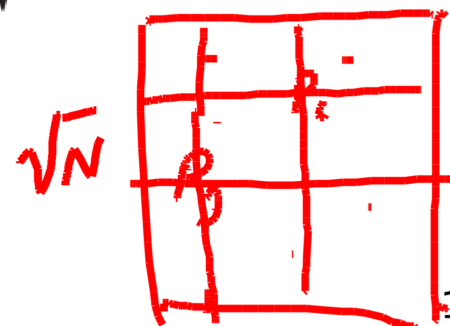
On receipt of a release from p_i at p_j

if (queue of requests is non-empty)
then
remove head of queue – from p_k, say;
send *reply* to p_k;
voted := TRUE;
else
voted := FALSE;
end if

© Pearson Education 2012

Maekawa's voting algorithm

- What is the optimal solution?
  - We should minimize K
  - $K \sim \sqrt{N}$ and $M = K$
- It is non-trivial to calculate the optimal sets R_i
  - approximation: place the processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix and let R_i be the union of the row and column containing p_i . $|R_i| \sim 2\sqrt{N}$



Maekawa's voting properties

- ✔ This algorithm achieves the safety property, ME1
  - If it were possible for two processes p_i and p_j to enter the critical section at the same time, then the processes in $V_i \cap V_j \neq \emptyset$ would have to have voted for both i and j
- ✗ Unfortunately, the algorithm is deadlock-prone
  - Consider three processes, p_1 , p_2, p_3 , with $V_1=\{p_1,p_2\}$, $V_2 =\{p_2, p_3\}$ and $V_3 = \{p_3, p_1\}$, If the three processes concurrently request entry to the critical section
    - p1 replies to itself and hold off p2 , p2 reply to itself and hold off p3 , and p3 reply to itself and hold off p1

Maekawa's voting properties

- We can achieve ME2 and ME3 through ordering the requests

Maekawa's voting performance

- bandwidth utilization is $2\sqrt{N}$ messages per entry to the critical section
- $\sqrt{N}$ messages per exit
- The total $3\sqrt{N}$ is less than the 2(N – 1) messages required by Ricart and Agrawala's (if N > 4)
- The client delay is the same as that of Ricart and Agrawala's algorithm
  - but the synchronization delay is worse: a round-trip time instead of a single message transmission time.

Election

Election in DS

- An algorithm for choosing a unique process to play a particular role is called an election algorithm
- Examples:
  - In central-server algorithm for mutual exclusion, the ‘server’ is elected from among the processes that need to use the critical section
  - Selecting a master between some replica in Google File System

Election applications

- Leader is useful for coordination among distributed servers
- Apache Zookeeper
  - a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services
- Google’s Chubby
  - Providing lock service for loosely-coupled distributed systems

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?
- Election algorithm 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 id.
- Messages are eventually delivered.
- Failures may occur during the election protocol.

Calling 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.

Election algorithm requirements

- 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) and for all non-faulty processes p: p’s elected is not Null

Election algorithm requirements

- 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 computation (lowest computational load), or most disk space, or most number of files, etc

Bully algorithm

- Allows processes to crash during an election, although it assumes that message delivery between processes is reliable
- The algorithm assumes that the system is synchronous
  - it uses timeouts to detect a process failure
- The bully algorithm, assumes that each process knows which processes have higher identifiers, and that it can communicate with all such processes

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

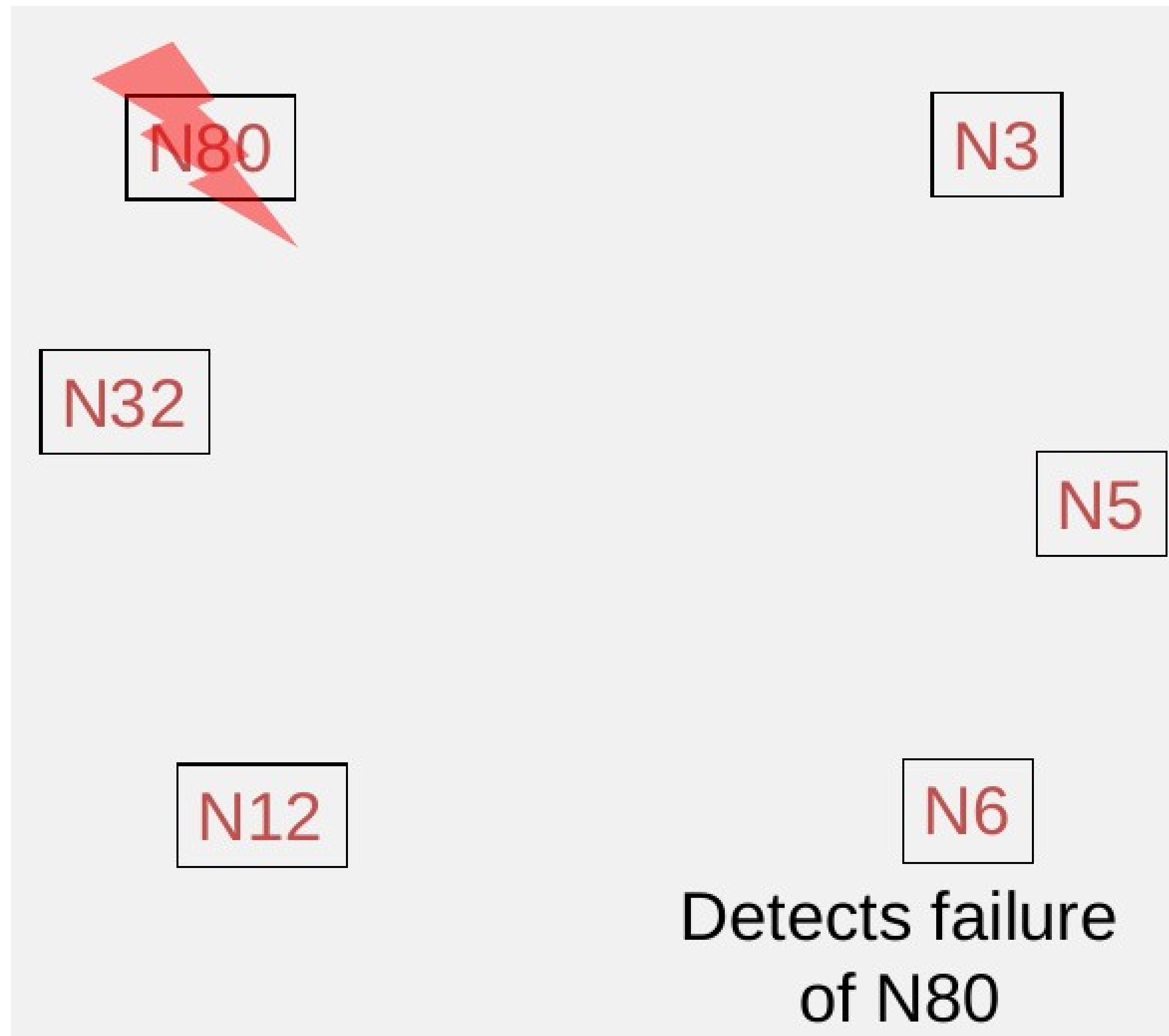
Bully algorithm

- 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.

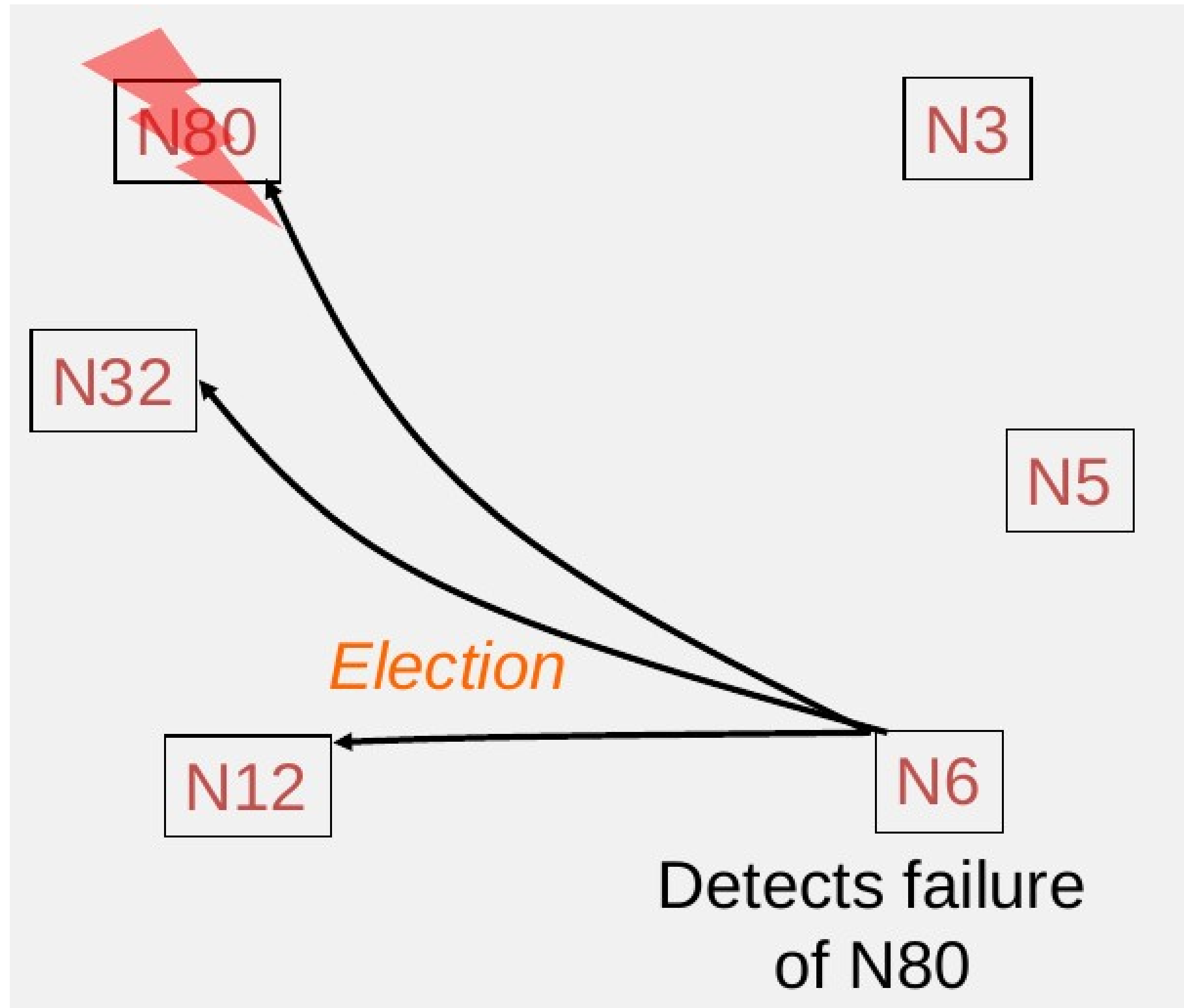
Bully algorithm

- A process that receives an Election message replies with OK message, and starts its own leader election protocol (unless it has already done so)

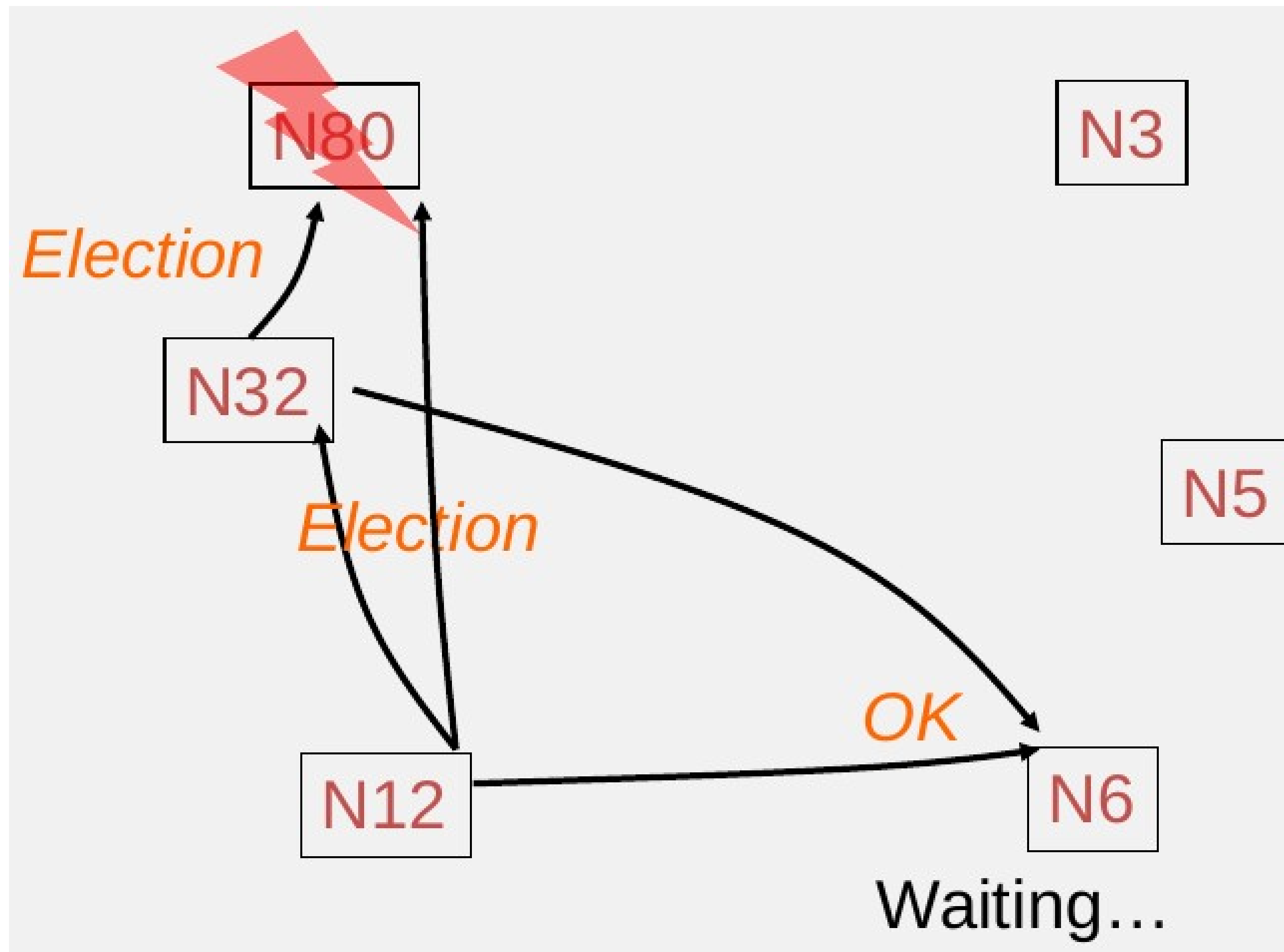
Bully algorithm Example



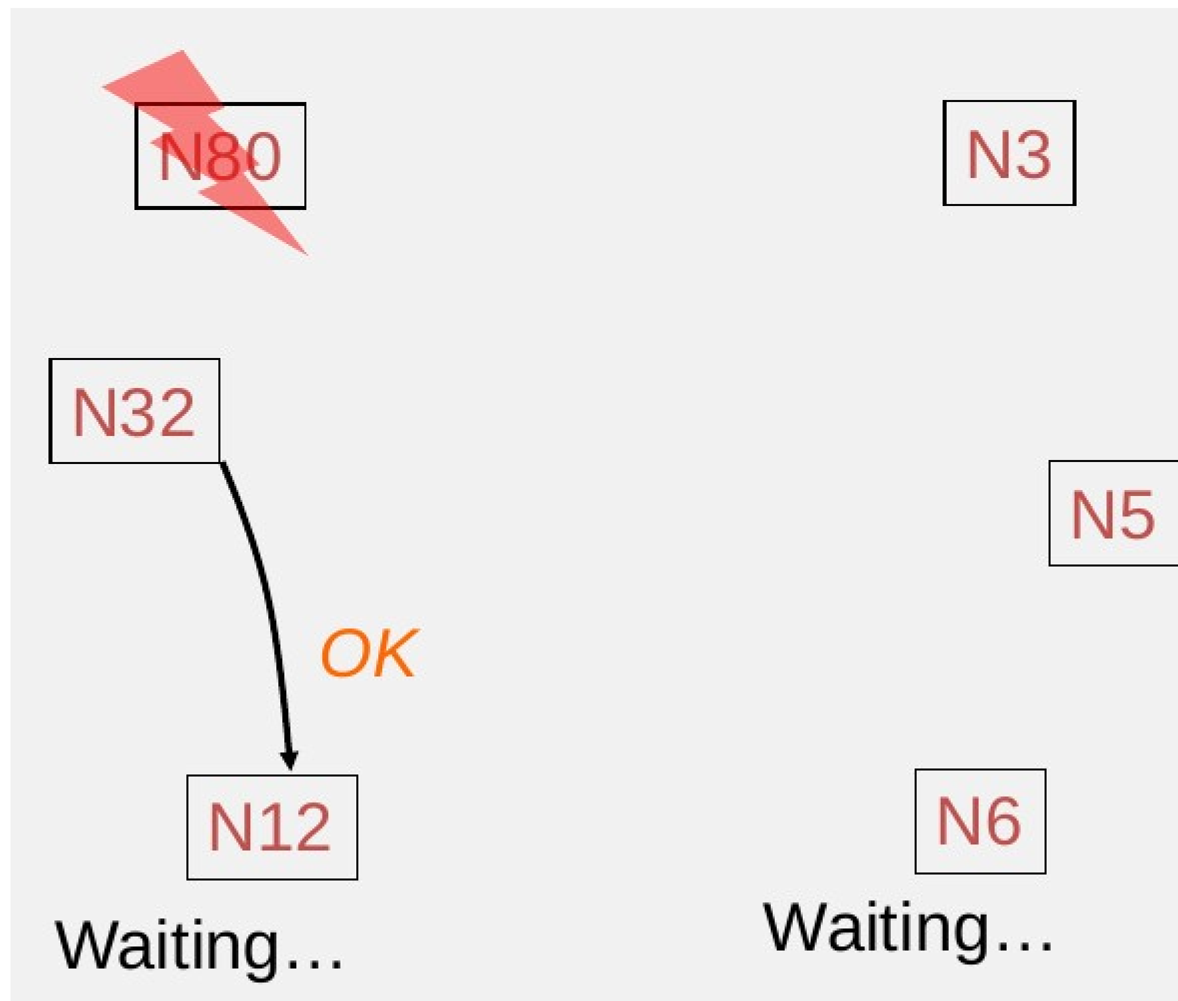
Bully algorithm Example



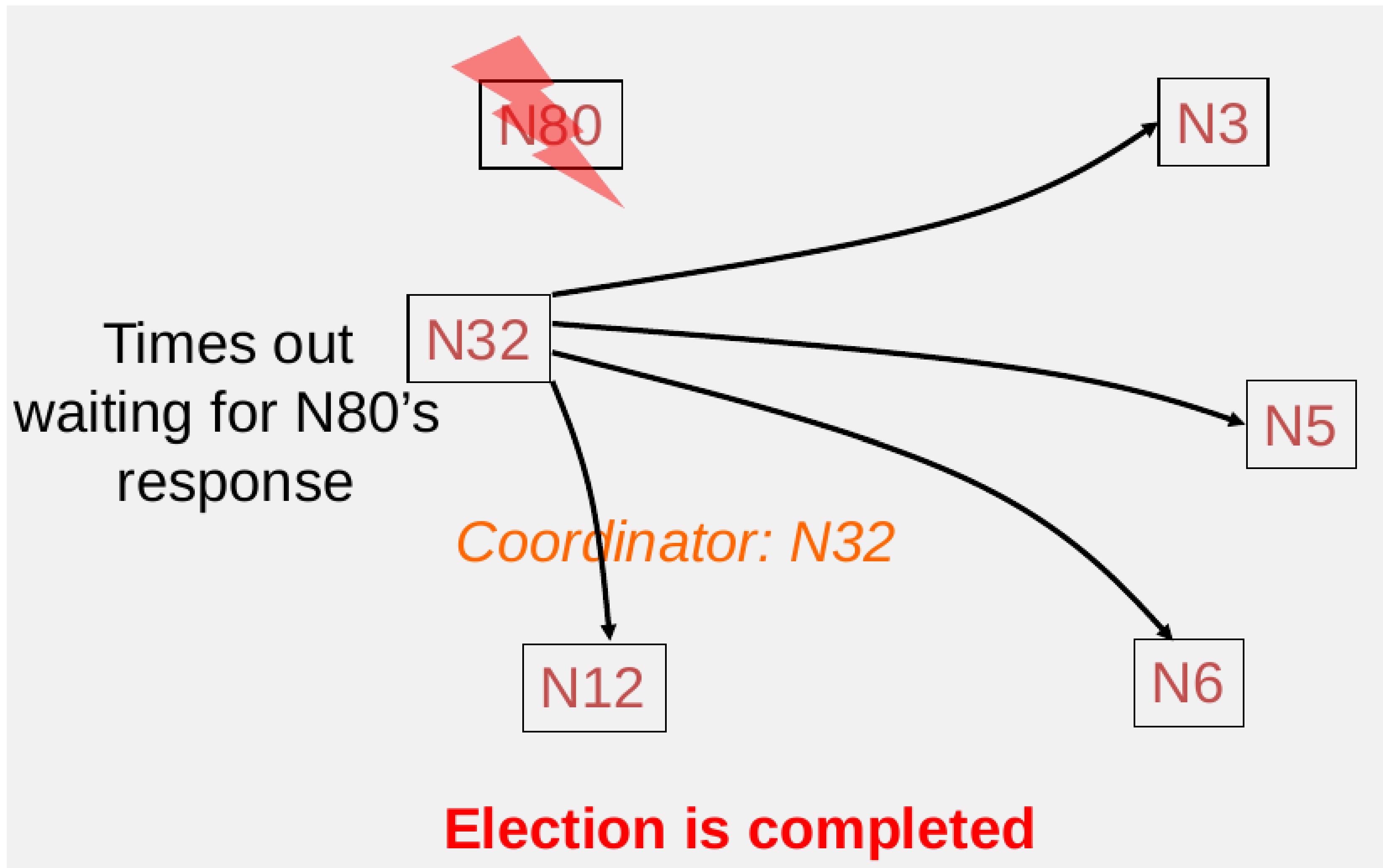
Bully algorithm Example



Bully algorithm Example

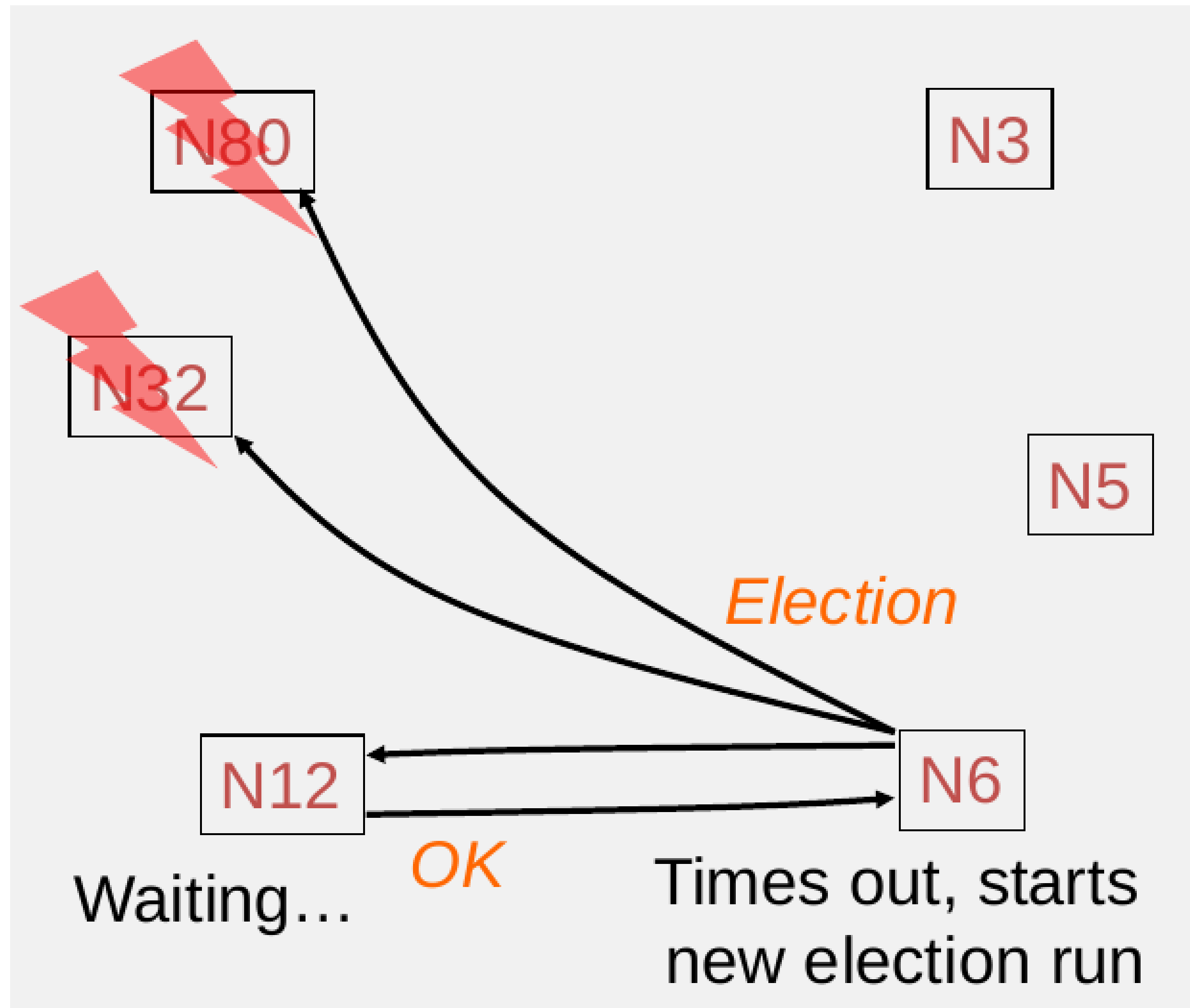


Bully algorithm Example

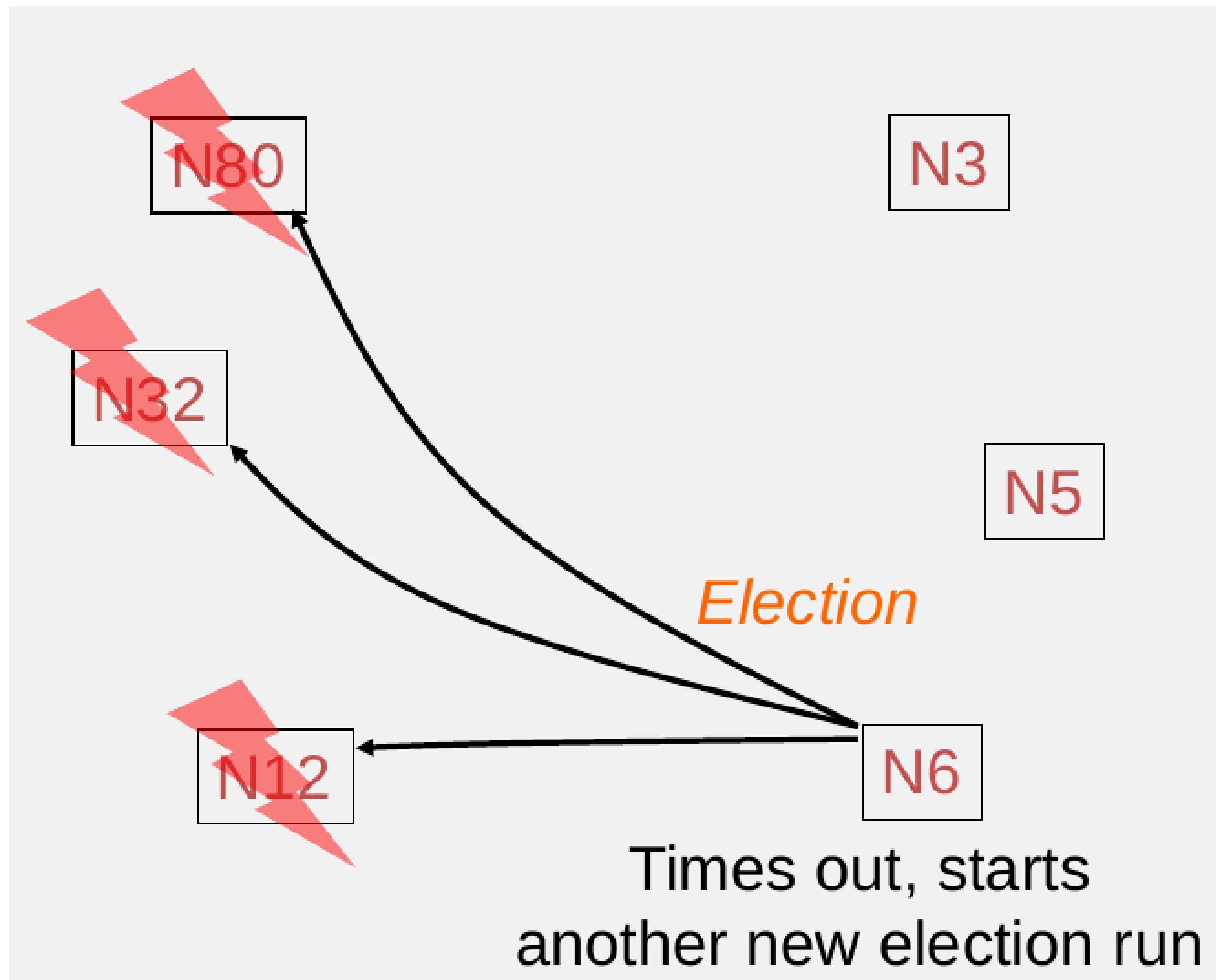


Failure during election run

Failure during election run



Failure during 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

    $= N-1 + N-2 + ... + 1 = (N-1)*N/2 = O(N^2)$

Analysis

- 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 latency can be calculated = worst-case process time + worst-case message latency

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 election

Consensus

What is the problem

- problems of agreement:
  - the problem is for processes to agree on a value after one or more of the processes has proposed what that value should be.

Three classic problem

- C: Consensus
- BG: Byzantine general problem
- IC: Interactive consistency
- Once we solve one of these problems, another ones can be solved through the first one

Solving one of 3 problems from one another's solution

- IC from BG
- BG from IC
- C from IC
- IC from C
- BG from C
- C from BG

Consensus Problem definition

- every process p_i begins in the undecided state and proposes a single value v_i , drawn from a set D $(i = 1, 2, ..., N)$.
- The processes communicate with one another, exchanging values.
- Each process then sets the value of a decision variable, d_i
- In doing so it enters the decided state, in which it may no longer change $d_i (i = 1, 2,.., N)$

Consensus solution requirement

- Termination: Eventually each correct process sets its decision variable.
- Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the decided state, then $d_i = d_j$ ($i, j = 1, 2,..,N$).
- Integrity: If the processes (correct or not) all proposed the same value, then any correct process in the decided state has chosen that value.

Consensus in a system with failure

- Consensus is possible to solve in a synchronous system where message delays and processing delays are bounded
- Consensus is impossible to solve in an asynchronous system where these delays are unbounded

Consensus in a synchronous system

- The algorithm uses only a basic multicast protocol.
- It assumes that up to f of the N processes exhibit crash failures.
- To reach consensus, each correct process collects proposed values from the other processes.
- The algorithm proceeds in $f + 1$ rounds, in each of which the correct processes multicast the values between themselves

Consensus in a synchronous system

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f + 1)$

B-multicast$(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

Proof (agreement and integrity after f+1 rounds)

Assume, to the contrary, that two processes differ in their final set of values. Without loss of generality, some correct process p_i possesses a value v that another correct process p_j $(i \neq j)$ does not possess. The only explanation for p_i possessing a proposed value v at the end that p_j does not possess is that any third process, p_k, say, that managed to send v to p_i crashed before v could be delivered to p_j. In turn, any process sending v in the previous round must have crashed, to explain why p_k possesses v in that round but p_j did not receive it. Proceeding in this way, we have to posit at least one crash in each of the preceding rounds. But we have assumed that at most f crashes can occur, and there are $f + 1$ rounds. We have arrived at a contradiction.

- The duration of a round is limited by setting a timeout based on the maximum time for a correct process to multicast a message.

Paxos

Paxos

- Paxos is an algorithm that is used to achieve consensus among a distributed set of computers that communicate via an asynchronous network.
- One or more clients proposes a value to Paxos and we have consensus when a majority of systems running Paxos agrees on one of the proposed values

Paxos liveness

- Paxos won’t try to specify precise liveness requirements.
- However, the goal is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value

Paxos application

- The most common use of Paxos is in implementing replicated machines, such as chunk servers in GFS
  - To ensure that replicas are consistent, incoming operations must be processed in the same order on all systems.
  - Each of the servers will maintain a log that is sequenced identically to the logs on the other servers.
  - A consensus algorithm will decide the next value that goes on the log.
  - Then, each server simply processes the log in order and applies the requested operations.

Paxos roles

- The nodes in paxos have three roles
  - Proposers
  - Acceptors
  - Learners
- Paxos nodes may take multiple roles, even all of them
- Paxos is a two phase algorithm

Phase Promise

- Proposer wants to propose a certain value
  - It sends prepare (IDp) to a majority or all of the acceptors (IDs must be unique)
    - If timeout, retry with a new one (a greater one)

```
ID = timestamp+pid;
send PREPARE(ID)
```

- Acceptor recieves a prepare message for IDp

```
Is this ID bigger than any round I have previously received?
If yes
    store the ID number, max_id = ID
    respond with a PROMISE message
If no
    do not respond (or respond with a "fail" message)
```

Phase Accept

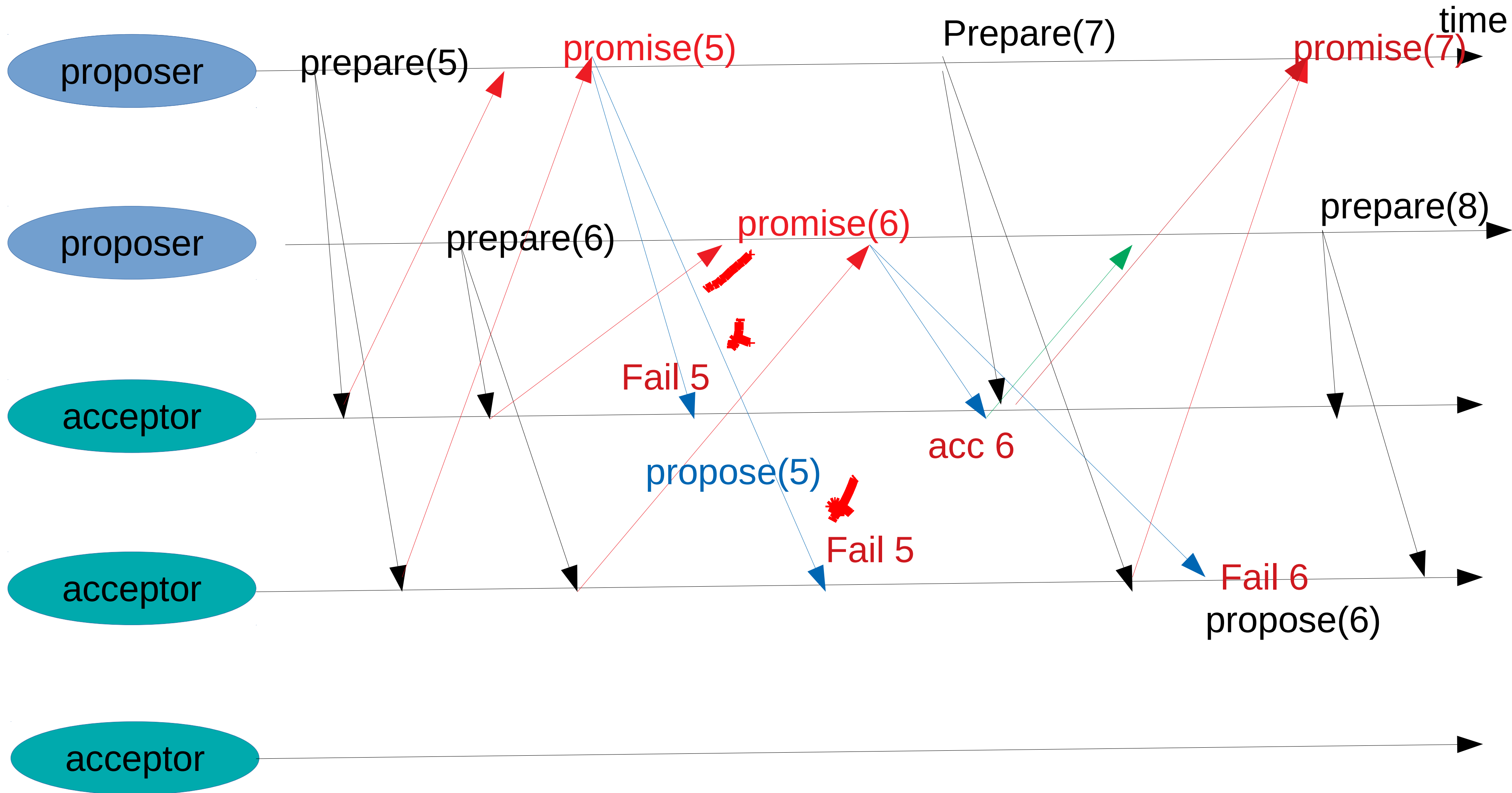
- If a proposer received a PROMISE message from the majority of acceptors, it now has to tell the acceptors to accept that proposal. If not, it has to start over with another round of Paxos.

  PROPOSE(ID, VALUE)

- The acceptor accepts the proposal if the ID number of the proposal is still the largest one that it has seen.

  Is the ID the largest I have seen so far, max_id == N?
  if yes
      reply with an ACCEPTED message & send ACCEPTED(ID, VALUE) to all learners
  if no
      do not respond (or respond with a "fail" message)

Contention



Fixing the protocol (I)

- acceptor receives a PREPARE(ID) message:

```
is this ID bigger than any round I have previously received?
if yes
    store the ID number, max_id = ID
    respond with a PROMISE(ID) message
if no
        did I already accept a proposal?
        if yes
            respond with a PROMISE(ID, accepted_ID,
            accepted_VALUE) message
        if no
            do not respond (or respond with a "fail" message)
```

Fixing the protocol (II)

- proposer receives PROMISE(ID, [VALUE]) messages:

```
do I have PROMISE responses from a majority of acceptors?
if yes
    do any responses contain accepted values (from other
    proposals)?
if yes
   pick the value with the highest accepted ID
   send PROPOSE(ID, accepted_VALUE) to at least a majority of
   acceptors
if no
   we can use our proposed value
   send PROPOSE(ID, VALUE) to at least a majority of acceptors
```

Failure analysis

- Suppose failure of each of the players in different phases and analyze how the algorithm handle it

Chubby

Chubby Goals

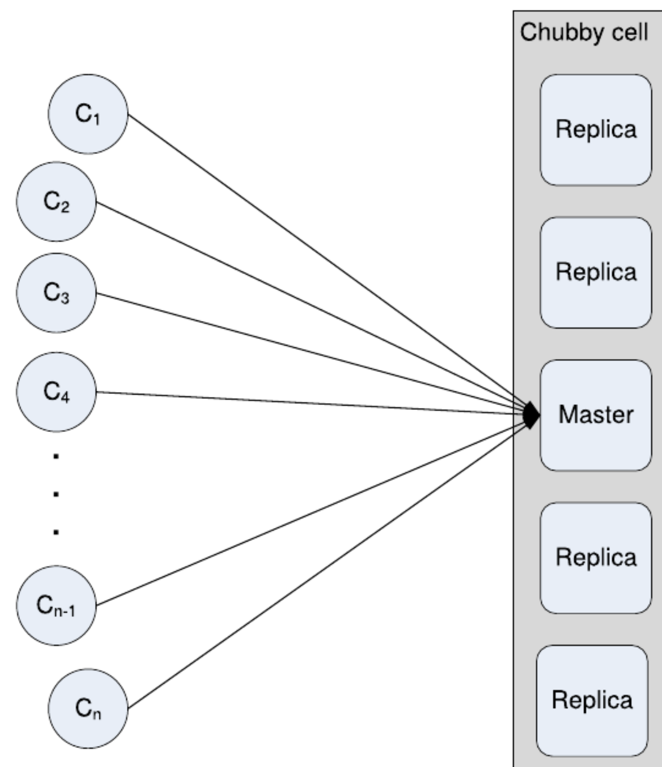
- Main Intention: Chubby is a distributed lock service intended for advisory coarse-grained synchronization of activities within Google's distributed systems;
- The primary goals: reliability, availability to a moderately large set of clients, and easy-to-understand semantics;
- The secondary goals: throughput and storage capacity

applications

- Google File System uses a Chubby lock to appoint a GFS master server
- Bigtable uses Chubby in several ways:
  - to elect a master
  - to allow the master to discover the servers it controls
  - to permit clients to find the master
- both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data;

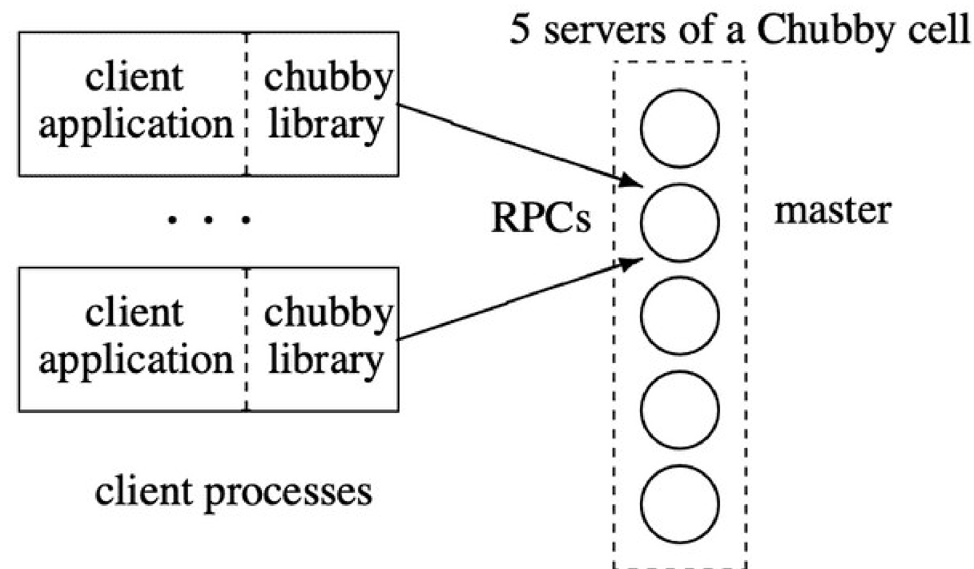
Chubby Design

- A Chubby cell consisting of some replicas (standard is 5), one of them is elected as the master. Clients c1 , c2 , . . . , cn communicate with the master using RPCs



Chubby Design

- Chubby replicas use asynchronous Paxos algorithm to elect a new master (master lease) when the current one fails
- Clients find the master by sending master location requests to the replicas listed in the DNS.



Chubby Design

- Clients use RPCs to request services from the master.
  - When a master receives a write request, it propagates the request to all replicas and waits for a reply from a majority of replicas before responding.
  - The master responds without consulting the replicas when receiving a read request.

Chubby components

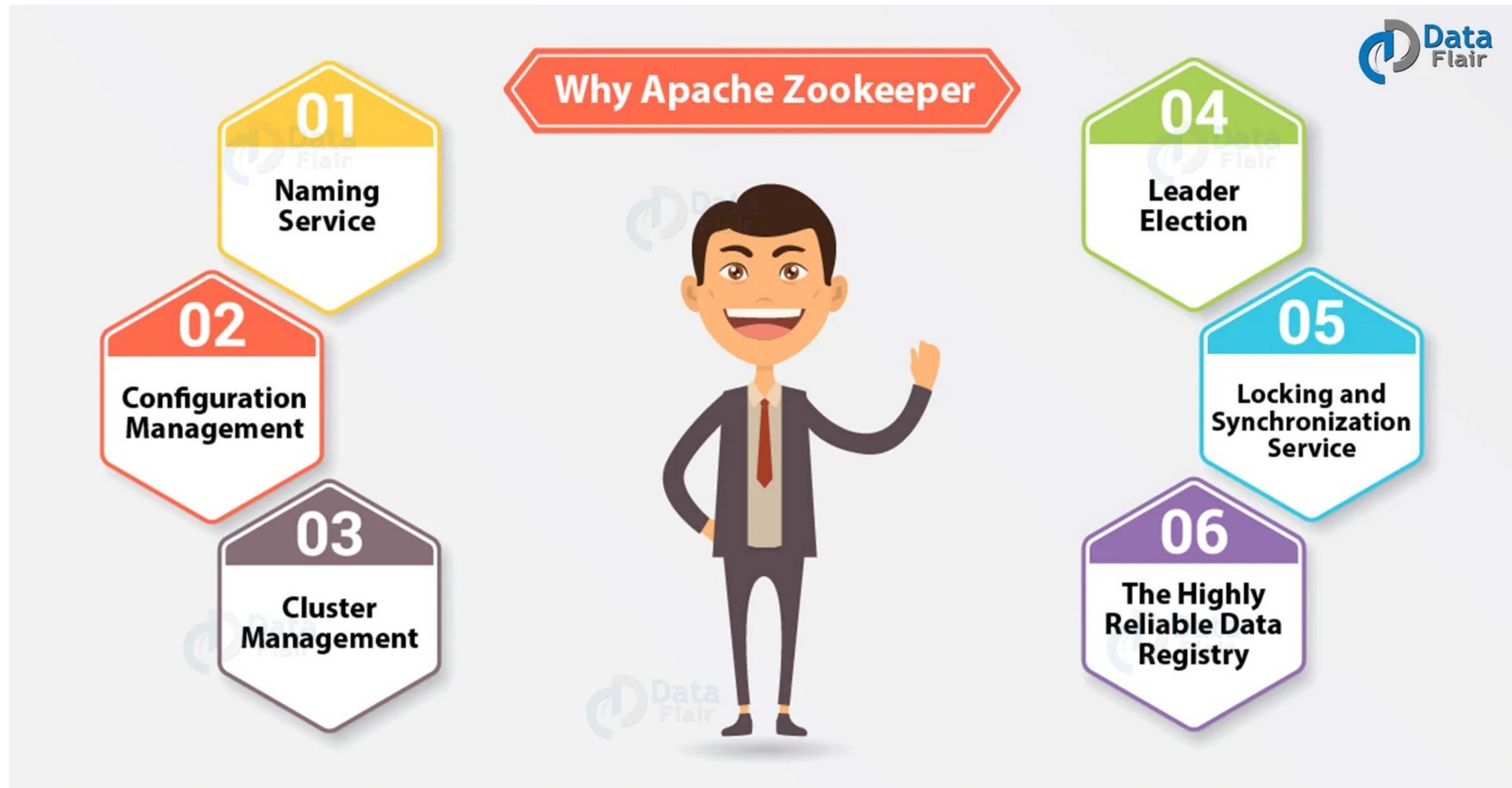
- Locks and Sequencers: Each Chubby file and directory can act as a reader-writer lock
  - either one client handle may hold the lock in exclusive (writer) mode, or any number of client handles may hold the lock in shared (reader) mode.
- API: Clients see a Chubby handle as a pointer to an opaque structure that supports various operations. Handles are created only by Open(), and destroyed with Close()

Zookeeper

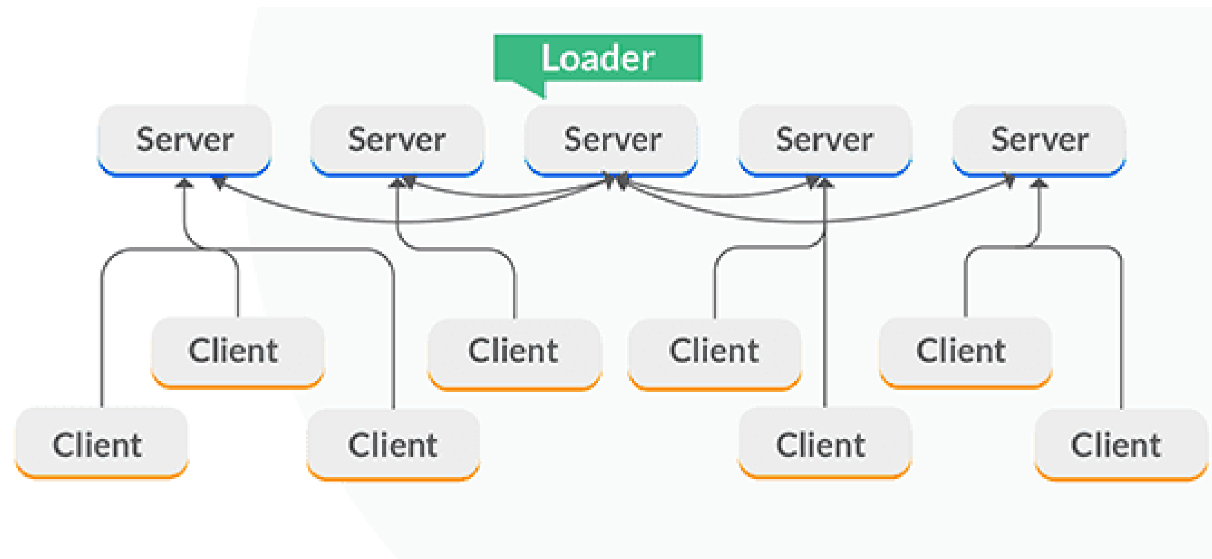
Zookeeper Goals

- ZooKeeper is a distributed coordination service with this goals:
  - Simplicity: With the help of a shared hierarchical namespace, it coordinates. it is organized as same as the standard file system with znodes.
  - Reliability: The system keeps performing, even if more than one node fails.
  - Speed: In the cases where ‘Reads’ are more common, it runs with the ratio of 10:1.
  - Scalability: By deploying more machines, the performance can be enhanced.

Zookeeper Goals



Zookeeper Architecture



Companies Using ZooKeeper

- Yahoo
- Hadoop and HBase
- Facebook
- eBay
- Twitter
- Netflix
- Zynga
- Nutanix