CES-27 Processamento Distribuído

Mutual Exclusion

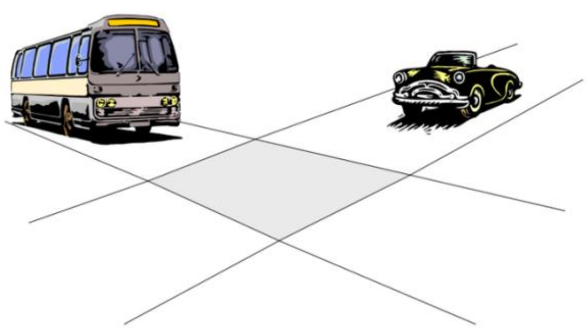
Prof Juliana Bezerra
Prof Celso Hirata
Prof Vitor Curtis

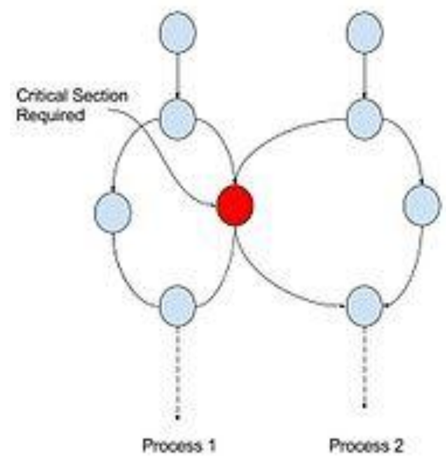
Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

What is mutual exclusion?

- Scenario
  - Multiple processes may want to **access the resource concurrently**
  - At any moment in time at most one process should be privileged ⇨ to have access
- Critical section (CS)
  - A block of source code where the process needs access to the resource, so it needs to be executed atomically
- Mutual exclusion
  - It aims to serialize access to a shared resource






```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

What is mutual exclusion?

- Properties of mutual exclusion algorithms
  - Safety (Mutual exclusion)
    - In every configuration, at most one process is privileged
  - Liveness (Starvation-freeness)
    - If a process *P* tries to enter its critical section, and no process remains privileged forever, then *P* will eventually enter its critical section.

Mutual exclusion in parallel system

- Remembering… two main concurrency models are **shared memory** and **message passing**

- Parallel systems is characterized by **shared memory**

- Atomicity is obtained by keeping a lock on the bus from the moment the value is read until the moment the new value is written
  - E.g. lock, semaphore, mutex, monitor
  - Potential problems: deadlocks

Mutual exclusion in distributed systems

- The core of distributed computation is **message passing**

- Classes of algorithms
  - Token-based algorithms
    - They use auxiliary resources such as **tokens** to resolve the conflicts
      - The process holding the token is privileged
    - Examples:
      - *Centralized algorithm
      - *Token ring algorithm
      - Raymond's algorithm

We will study algorithms marked with *

  - Timestamp-based algorithms
    - They resolve conflict in use of resources based on **timestamps** assigned to requests of resources.
      - Requests for entering a critical section are prioritized by means of logical timestamps
    - Examples:
      - Lamport's algorithm
      - *Ricart-Agrawala

  - Quorum-based algorithms
    - To become privileged, a process needs the permission from a quorum of processes
    - Examples:
      - *Maekawa
      - Agrawal-El Abbadi algorithm

Mutual exclusion in distributed systems

- System model (our assumptions)
  - Each pair of processes is connected by reliable channels
  - Messages are eventually delivered to recipient, and in FIFO order
  - Processes do not fail
    - Fault-tolerant variants exist in literature

- General directives about performance
  - Efficient algorithms use **fewer messages** and make processes **wait for short** durations to access CS
  - Metrics:
    - Bandwidth
      - The total number of messages sent in each *enter* and *exit* operation
    - Client delay
      - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
    - Synchronization delay
      - The time interval between one process exists CS and next process enters, when only one process is waiting

Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

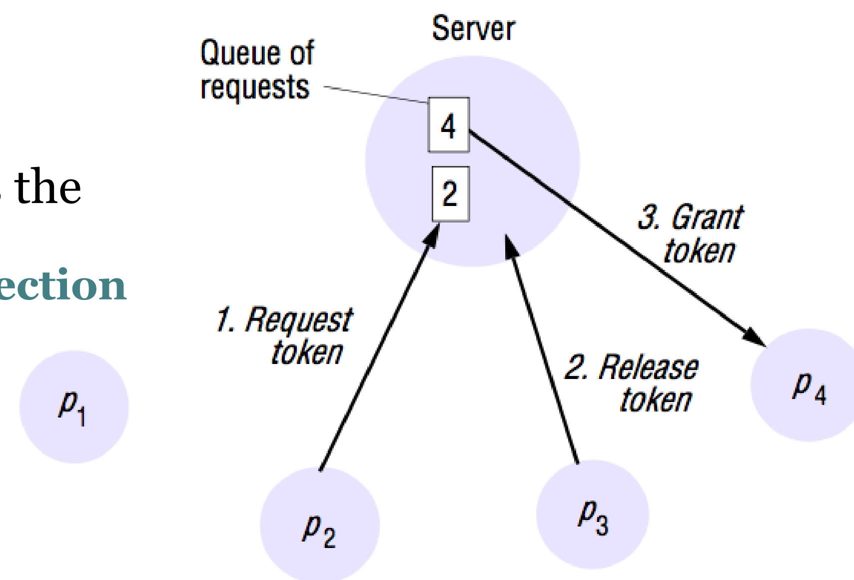
Centralized algorithm

- Mimic single processor system
- The process, that has the token, has the resource!

- One process works as **coordinator** (**master/leader/server**) that controls the granting of the token
  - Coordinator is chosen using one of our **election algorithms**!

- Other processes can:
  - Request resource
  - Wait for response
  - Receive grant
  - Access resource
  - Release resource

- If a process claims the resource, and the resource is hold by other process, the coordinator:
  - Does not reply until release
  - Maintains the request in its queue (FIFO order)



Note: 4 is in the top of the queue

Centralized algorithm

- Coordinator actions
  - On receiving a request from process P*i*

    if (master has token)

        Send token to P*i*

    else

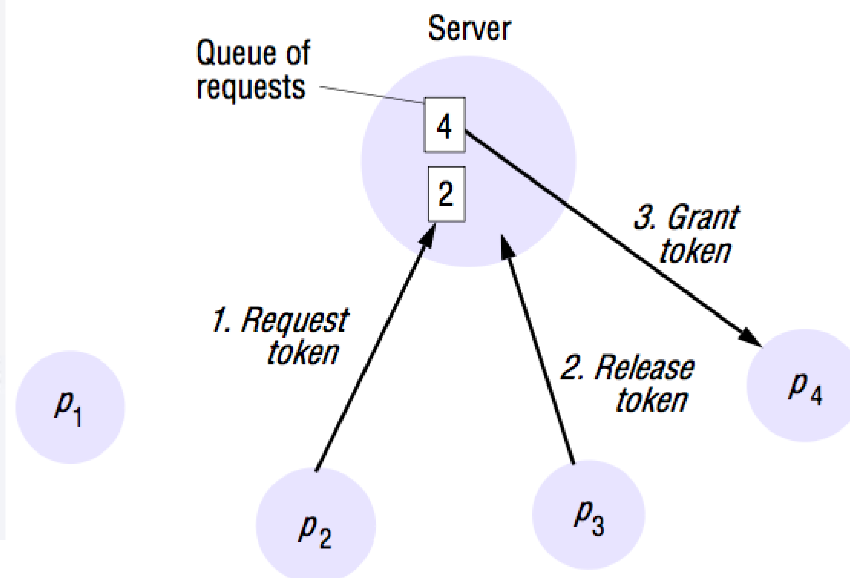
        Add P*i* to queue
  - On receiving a token from process P*i*

    if (queue is not empty)

        Dequeue head of queue (say P*j*), send that process the token

    else

        Retain token



- Benefits
  - Easy to implement, understand, verify
  - Safety
    - At most one process in CS (one token)
  - Liveness
    - All requests processed in order
    - No process waits forever
    - Every request for CS granted eventually

Note: 4 is in the top of the queue

Centralized algorithm

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

Metric	Centralized	Token ring	Ricart-Agrawala	Maekawa
Bandwidth	*Enter*: 2 messages *Exit*: 1 message Then **O(1)**			
Client delay	2 messages latencies (request + grant) Then **O(1)**			
Synch. delay	2 messages latencies (release + grant) Then **O(1)**			

Centralized algorithm

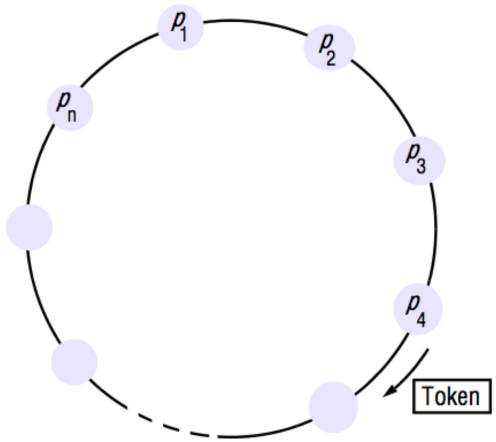
- Problems
  - The coordinator is a single point of failure
    - If it crashes, the entire system may go down
  - If processes normally block after making a request, they cannot distinguish a **dead coordinator** from "**access denied**" since in both cases coordinator does not reply
  - In a large system, a single coordinator has to take care of all process
    - The coordinator can be a **bottleneck**
  - Multiple resources can lead to a deadlock!

Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

Token ring algorithm

- Consider a number of processes that wish to enter in a critical section



- Assumptions
  - Unidirectional logical ring of processes
  - No duplication or message corruption
  - Possible loss of message

- Safety
  - Only one token
  - Exclusion is guaranteed by allowing one process to enter the critical section if and only if it has the token

- Liveness
  - In order to avoid starvation, the token circulates around the sites

Token ring algorithm

ALGORITHM: (processes **0 to n-1**)

All **n processes** executing the **same algorithm (textual symmetry)**

Process P_i:

...

receive token **from** P(n+i-1) mod n

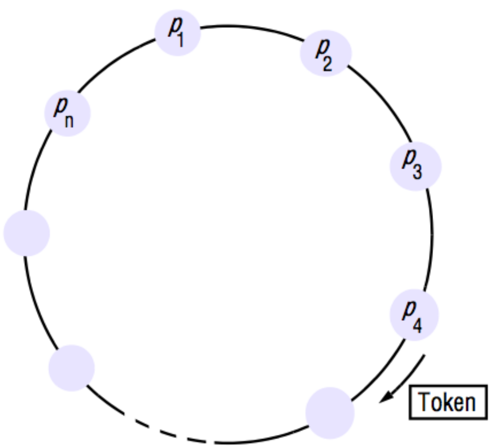
Receive the **token** from the **previous process** (**clockwise**)

<critical section>

send token **to** P(i+1) mod n;

Not necessarily the **process** enters the **critical section**. If doesn't, just **forwards** the **token** (**clockwise**)

Sends the token to the **next** process in the ring (**clockwise**)



Token ring algorithm

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

Metric	Centralized	Token ring	Ricart-Agrawala	Maekawa
Bandwidth	*Enter*: 2 messages *Exit*: 1 message Then **O(1)**	*Enter*: N messages through the ring *Exit*: 1 message Then **O(N)**		
Client delay	2 messages latencies (request + grant) Then **O(1)**	Best case: already have the token ⇨ 0 messages Worst case: just sent token to neighbor ⇨N messages Then **O(N)**		
Synch. delay	2 messages latencies (release + grant) Then **O(1)**	Best case: process in *enter* is successor of process in *exit* ⇨ 1 message Worst case: process in *enter* is predecessor of process in *exit* ⇨ (N-1) messages Then **O(N)**		

Token ring algorithm

- Benefits
  - The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a CS
  - Since the token circulates among processes in a well-defined order, starvation cannot occur

- Problems
  - Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter
    - **Performance** (worst case)
      - Maximum delay = (N-1) (max[Critical Section] + overhead)
  - What happens on **process failure**?
    - **Ring re-connections** or re-establishment of local state variables needed
    - Failure detection is, usually, external to the processes
      - The observer must be in the center of the ring
  - What happens if the **token is lost**?
    - A new one must be generated
    - A possible problem: multiple token generation
    - Token loss detection and regeneration
      - Timeout-based algorithm
      - Misra algorithm (Ping-pong algorithm)

Attention: The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.

More details? See extra slides

Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

Ricart-Agrawala algorithm

- Reference: Glenn Ricart and Ashok K. Agrawala. "**An optimal algorithm for mutual exclusion in computer networks**." *Communications of the ACM* 24.1 (1981): 9-17.
- Classical algorithm from 1981
- No token ⇨use timestamp (Lamport logical time)
- Use the notion of causality and multicast

Ricart-Agrawala algorithm

Important:

- Every *request* sent by p_i has <T, p_i>
- A *reply* sent by p_i has <T_i, p_i>

T := time when p_i requested CS
T_i := current clock of p_i
p_i := process id

- In order to be prepared to next requests of CS, a process should update its clock after receiving any message. You can use Lamport idea: 1+ maximum (my clock, received clock)

```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast request to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;
On receipt of a request <T_i, p_i> at p_j (i ≤ j)
    if  (state = HELD or (state = WANTED and (T, p_j) < (T_i, p_i)))
    then
        queue request from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```


T = Time when I requested CS

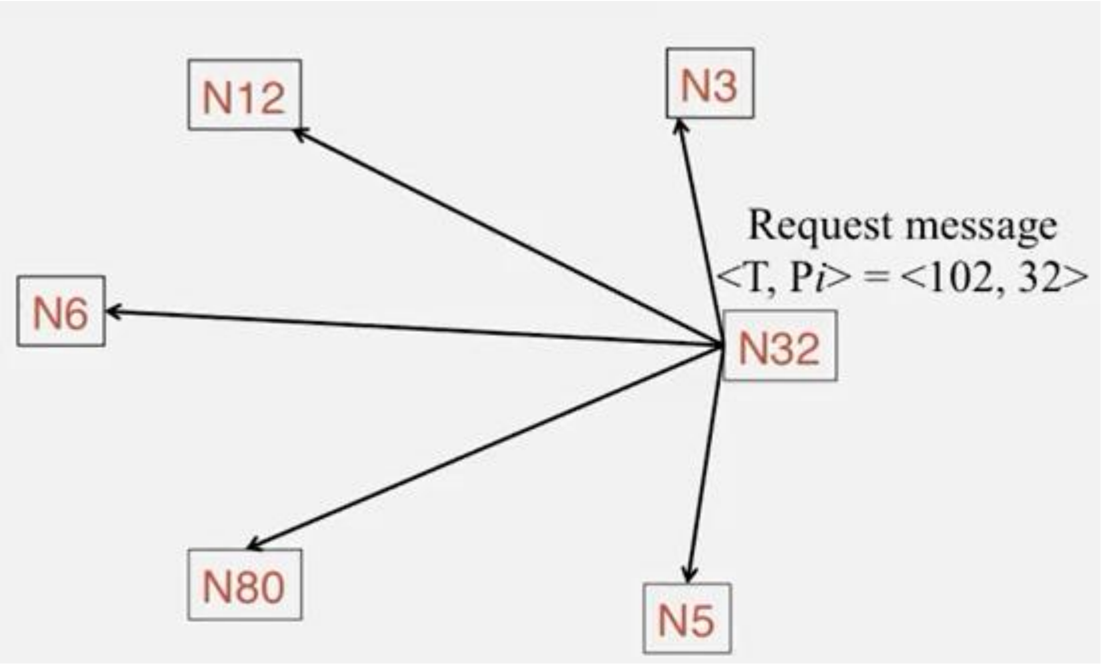
Waiting in a non-blocking mode, so I can process other messages

I am in CS

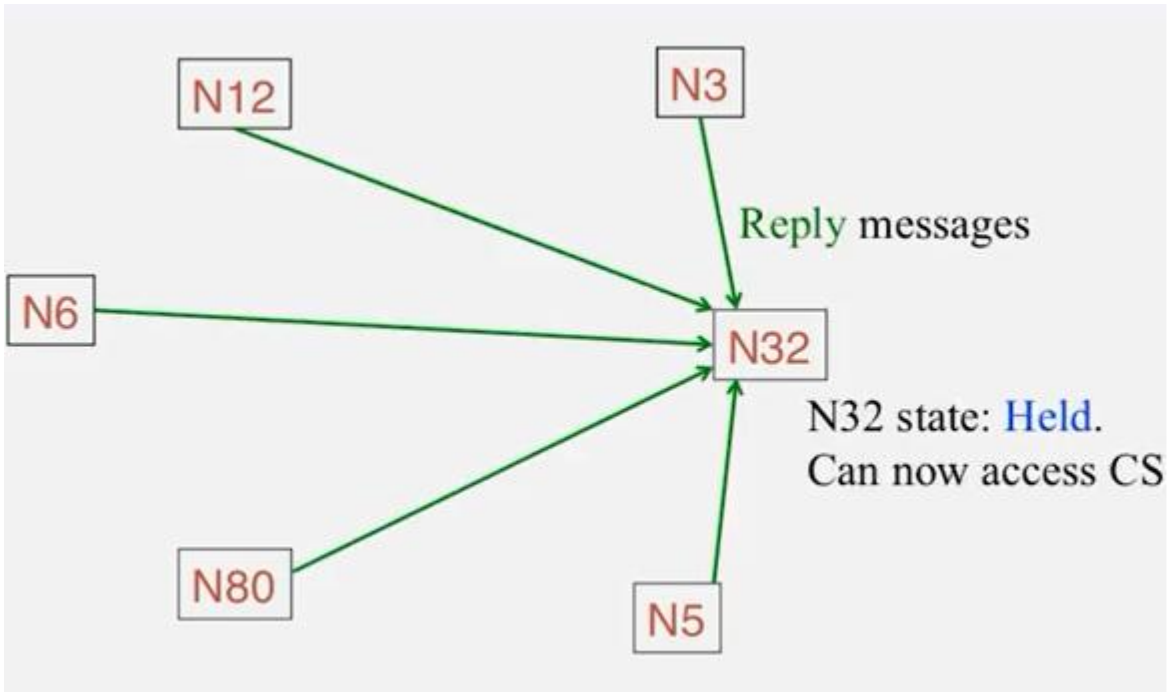
I want CS and the preference is mine!

If T==Ti, the preference is for the lower process id (in this case i, since i<=j)

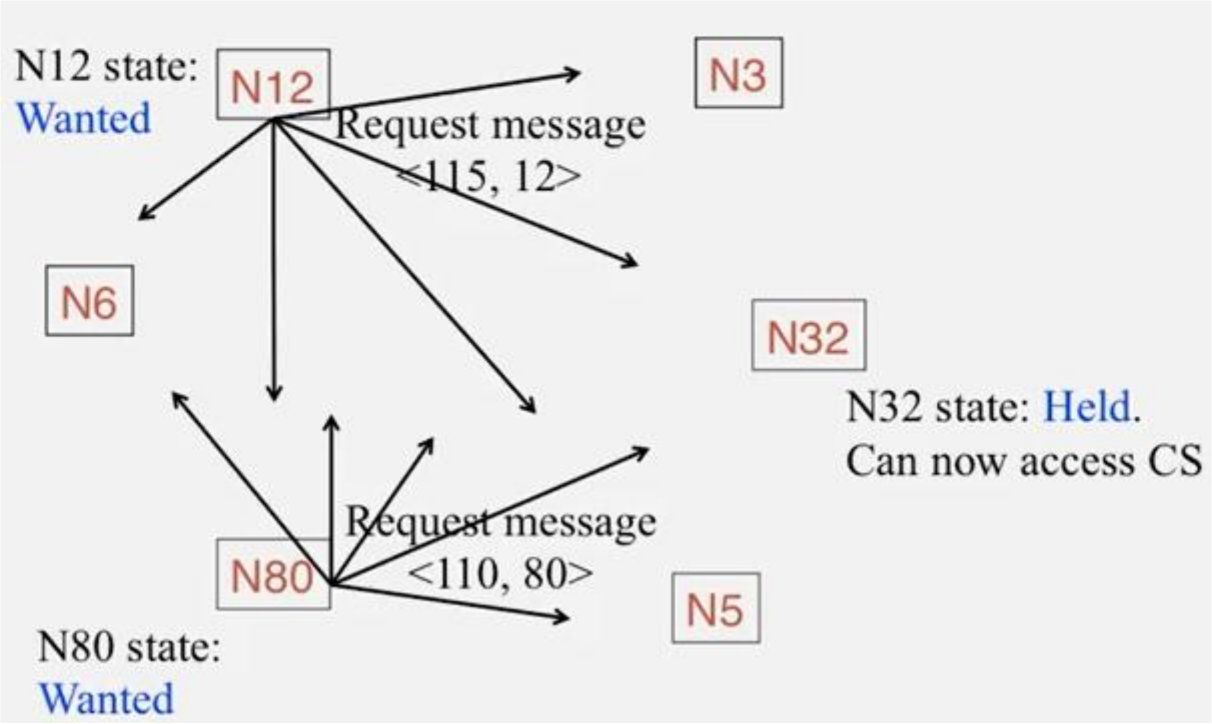
Ricart-Agrawala algorithm



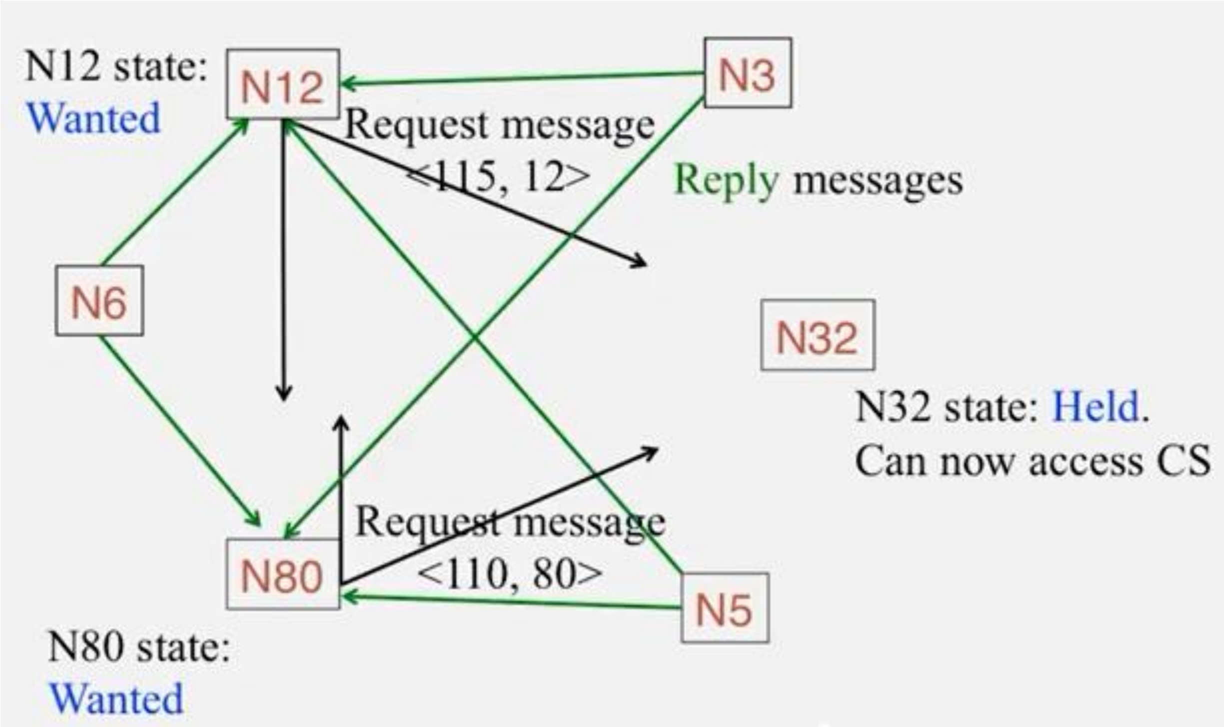
Ricart-Agrawala algorithm



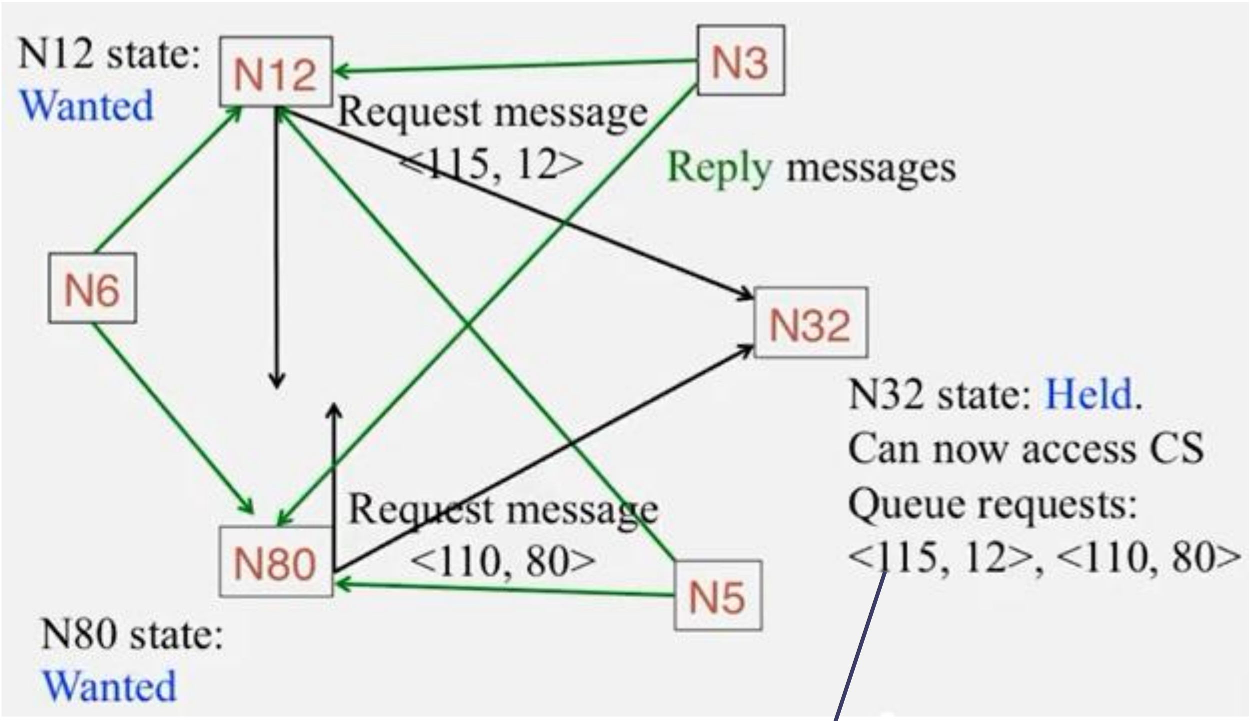
Ricart-Agrawala algorithm



Ricart-Agrawala algorithm

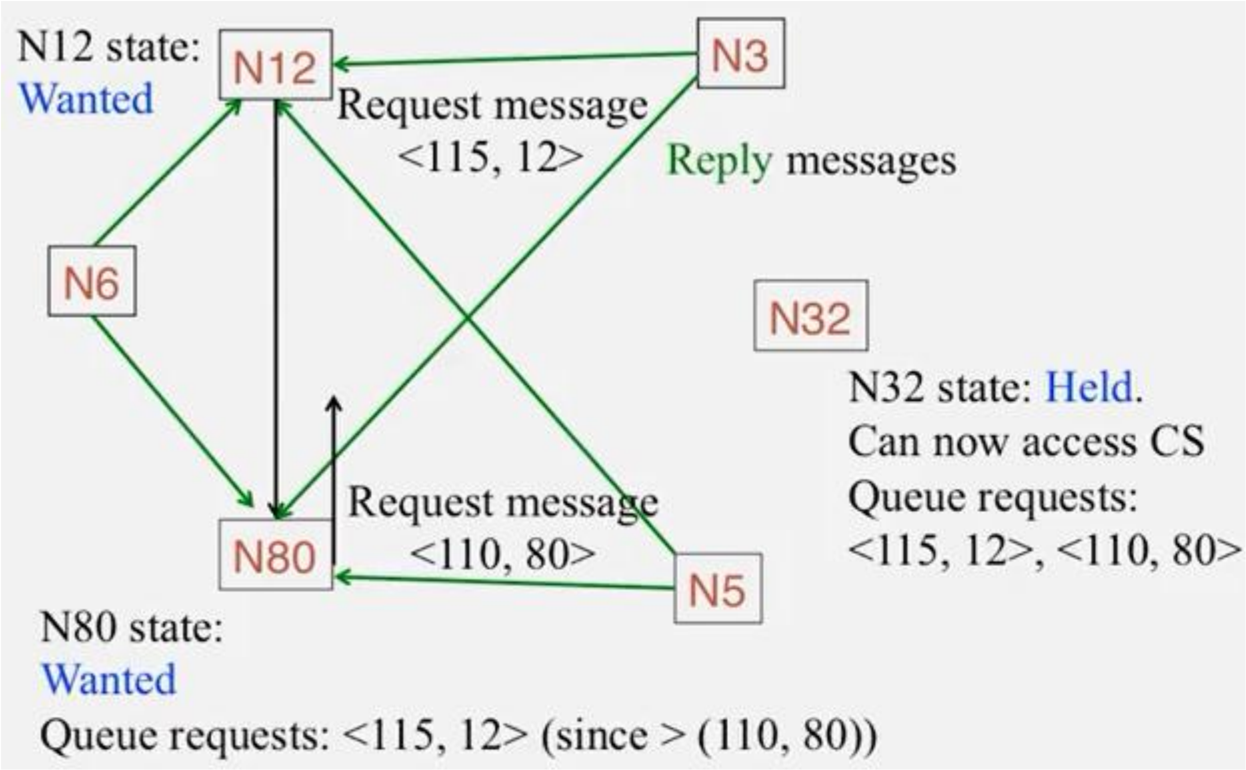


Ricart-Agrawala algorithm

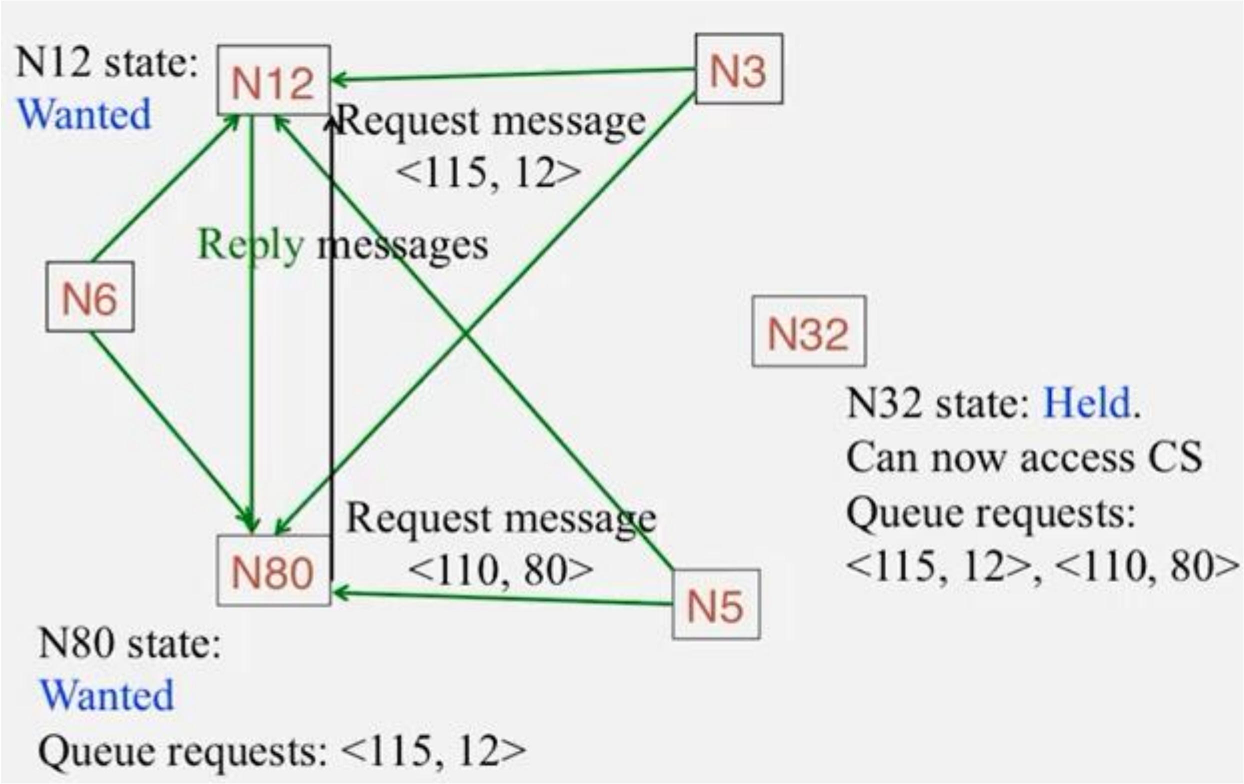


The order in queue here does not matter, since N32 will send *reply* to entire queue when exits CS

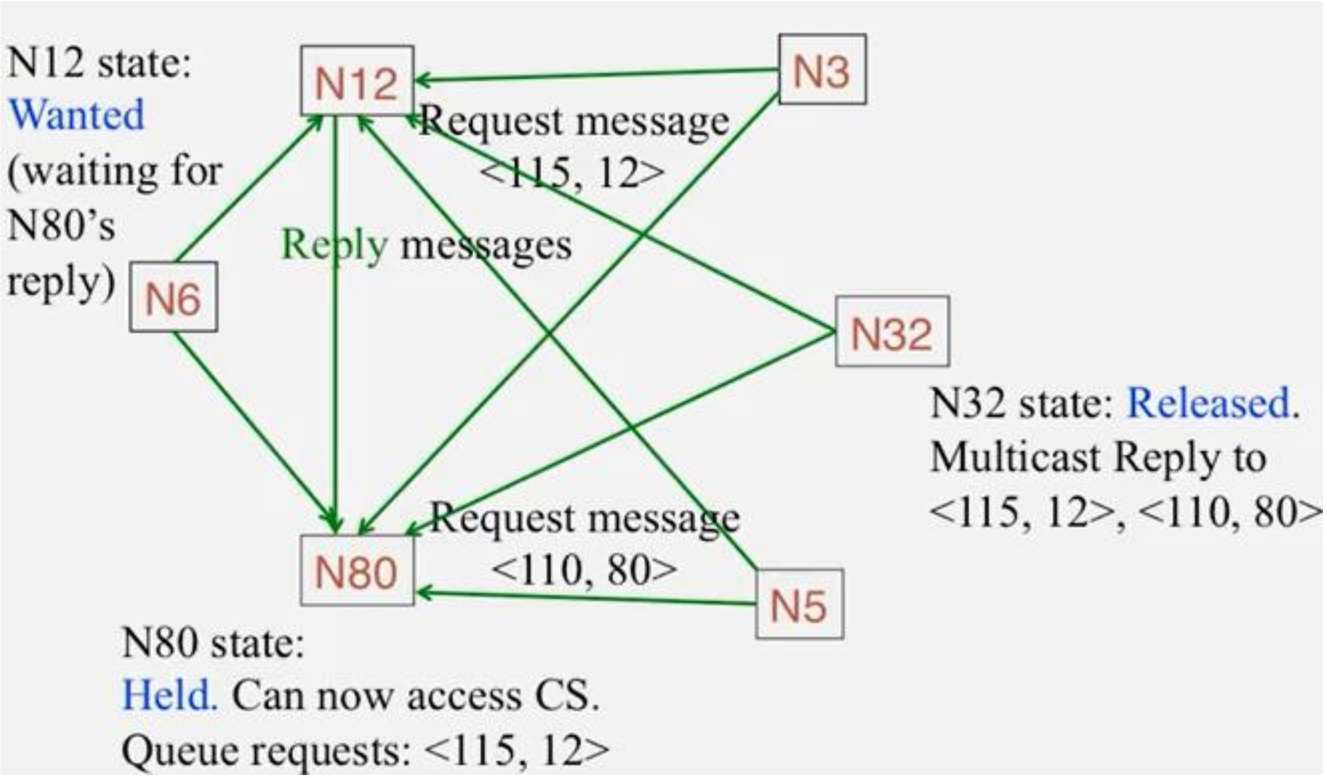
Ricart-Agrawala algorithm



Ricart-Agrawala algorithm



Ricart-Agrawala algorithm



Ricart-Agrawala algorithm

- Safety
  - Two processes p_i and p_j cannot both have access to CS
    - If they did, then both have sent *reply* to each other
    - Thus $(T_i, p_i) < (T_j, p_j)$ and $(T_j, p_j) < (T_i, p_i)$, which are together impossible
    - Imagine: If $(T_i, p_i) < (T_j, p_j)$ and p_i replies to p_j's request before creating its own request
      - It seems that p_i and p_j would approve each others' requests
      - But causality and Lamport timestamps at p_i implies that $T_i > T_j \Rightarrow$ contradiction
        - Since p_i updated its clock with p_j's request
      - Impossible situation!

- Liveness
  - Worst case: all other processes request CS, so wait for all (N-1) replies
  - But you will have CS eventually!

- Ordering
  - Requests with lower Lamport timestamps are granted earlier

Ricart-Agrawala algorithm

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

Metric	Centralized	Token ring	Ricart-Agrawala	Maekawa
Bandwidth	*Enter*: 2 messages *Exit*: 1 message Then **O(1)**	*Enter*: N messages through the ring *Exit*: 1 message Then **O(N)**	*Enter*: 2(N-1) messages *Exit*: (N-1) messages Then **O(N)**	I can do it better!
Client delay	2 messages latencies (request + grant) Then **O(1)**	Best case: already have the token ⇨ 0 messages Worst case: just sent token to neighbor ⇨N messages Then **O(N)**	Multicast (N-1) requests is O(1) + Receive (N-1) replies in parallel is O(1) Then **O(1)**	
Synch. delay	2 messages latencies (release + grant) Then **O(1)**	Best case: process in *enter* is successor of process in *exit* ⇨ 1 message Worst case: process in *enter* is predecessor of process in *exit* ⇨ (N-1) messages Then **O(N)**	1 reply from the process in CS Then **O(1)**	

Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

Maekawa algorithm

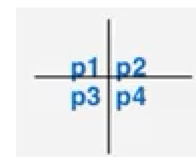
- Approach
  - To get access, **not all processes** have to agree
  - Suffices to split set of processes up into **subsets** ("***voting sets***") that **overlap**
    - Concept of **quorums**!
  - Suffices that there is consensus within every subset

- Key differences from Ricart-Agrawala
  - Each process requests permission from only its voting set members
    - Not from all as in Ricart-Agrawala
  - Each process (in a voting set) gives permission to at most one process at a time
    - Not to all as in Ricart-Agrawala

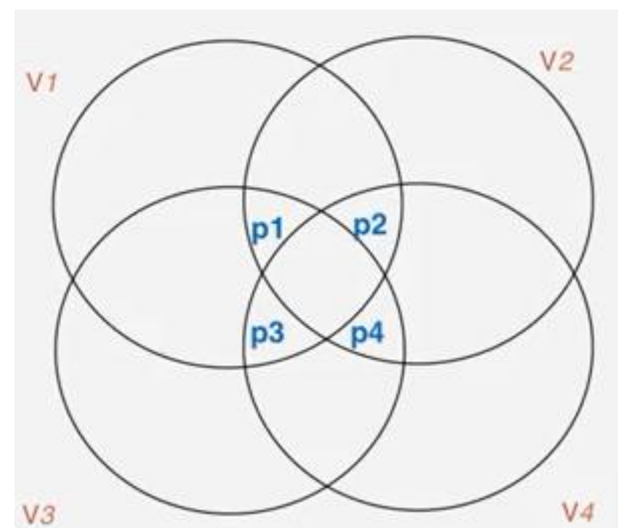
- Model
  - Processes p_1, .., p_N
  - Voting sets V_1, .., V_N chosen such that $\forall$ i,j and for some integer M:
    - $p_i \in V_i$ (I am in my voting set)
    - $V_i \cap V_j \neq \emptyset$ (some overlap in every voting set)
    - $|V_i| = K$ (fairness: all voting sets have equal size)
    - Each process p_k is contained in M voting sets

Maekawa algorithm

- Remembering the model…
  - Processes $p_1, .., p_N$
  - Voting sets $V_1, .., V_N$ chosen such that $\forall$ i,j and for some integer M:
    - $p_i \in V_i$ (I am in my voting set)
    - $V_i \cap V_j \neq \emptyset$ (some overlap in every voting set)
    - $|V_i| = K$ (fairness: all voting sets have equal size)
    - Each process p_k is contained in M voting sets



- Optimization goal
  - Minimize K while achieving mutual exclusion
    - It can be shown to be reached when $K \sim \sqrt{N}$ and M=K
  - Optimal voting sets: nontrivial to calculate
    - Approximation: derive V_i so that $|\mathbf{V_i}| \sim \mathbf{2\sqrt{N}}$
    - Place processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix
    - Let V_i the union of the row and column containing p_i



Maekawa 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 - {p_i};
    Wait until (number of replies received = (K - 1));
    state := HELD;
On receipt of a request from p_i at p_j (i ≠ 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 - {p_i};
On receipt of a release from p_i at p_j (i ≠ 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
```

Waiting in a non-blocking mode, so I can process other messages

Someone else is in CS, so I can not allow you to enter now

We need to send release to all Vi
And then include i=j here too.

Reference: Kshemkalyani & Singhal (2008)

Maekawa algorithm

- Safety
  - If possible for two processes to enter CS, then processes in the non-empty intersection of their voting sets would have granted access to both
  - Impossible, since all processes make at most one vote after receiving request

- Liveness
  - A process needs to wait for at most (N-1) other processes to finish CS
  - It does not guarantee liveness, since deadlocks are possible. E.g:
    - Three processes p_1, p_2 and p_3
    - $V_1 = V_2 = V_3 = \{p_1, p_2, p_3\}$
    - All processes requested CS
    - Possible to construct cyclic wait graph
      - p_1 replies to p_2, but queues request from p_3
      - p_2 replies to p_3, but queues request from p_1
      - p_3 replies to p_1, but queues request from p_2
  - There are deadlock-free version of the algorithm
    - Use of logical clocks
    - Processes queue requests in happened-before order

Maekawa algorithm

Note: Consider optimization goal So $|V_i| \sim \sqrt{N}$

- Remembering analysis metrics…
  - Bandwidth
    - The total number of messages sent in each *enter* CS and *exit* CS operation
  - Client delay
    - The delay incurred by a process at each *enter* (or *exit*) operation, when no process is in or waiting CS
  - Synchronization delay
    - The time interval between one process exists CS and next process enters, when only one process is waiting

Metric	Centralized	Token ring	Ricart-Agrawala	Maekawa
Bandwidth	*Enter*: 2 messages *Exit*: 1 message Then **O(1)**	*Enter*: N messages through the ring *Exit*: 1 message Then **O(N)**	*Enter*: 2(N-1) messages *Exit*: (N-1) messages Then **O(N)**	*Enter*: 2√N messages *Exit*: √N messages Then **O(√N)**
Client delay	2 messages latencies (request + grant) Then **O(1)**	Best case: already have the token ⇨ 0 messages Worst case: just sent token to neighbor ⇨N messages Then **O(N)**	Multicast (N-1) requests is O(1) + Receive (N-1) replies in parallel is O(1) Then **O(1)**	Multicast √ N requests is O(1) + Receive √ N replies in parallel is O(1) Then **O(1)**
Synch. delay	2 messages latencies (release + grant) Then **O(1)**	Best case: process in *enter* is successor of process in *exit* ⇨ 1 message Worst case: process in *enter* is predecessor of process in *exit* ⇨ (N-1) messages Then **O(N)**	1 reply from the process in CS Then **O(1)**	1 release from the process that exits CS + 1 reply from a process in the voting set (this process is common in the other voting set) Then **O(1)**

Mutual Exclusion Algorithms

- Notes on Fault Tolerance
  - None of these algorithms tolerates message loss
  - Token ring algorithm cannot tolerate single crash failure
  - Maekawa's algorithm can tolerate some crash failure
    - If process is in a voting set not required, rest of the system not affected
  - Centralized algorithm tolerates crash failure of node that has neither requested access nor is currently in the CS
  - Ricart-Agrawala algorithm can be modified to tolerate crash failures by the assumption that a failed process sends all replies immediately
    - It requires reliable failure detector

Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

Chubby

- Reference: Burrows, M. **The Chubby lock service for loosely-coupled distributed systems**. Google Inc. **2006**

- Chubby is a **distributed lock service** intended for **synchronization of activities** within Google's distributed systems

- The design emphasis is on **availability** and **reliability**, as <u>opposed to high performance</u>
  - The purpose of the lock service is to allow its **clients to synchronize their activities** and to **agree on basic information** about their environment

Chubby use in Google

- It is a common rendezvous mechanism for systems such as **MapReduce**
- The storage systems **GFS** (Google File System) and **Bigtable** use Chubby to elect a primary ("chunk master") from redundant replicas

```
x = Open("/ls/gfs-cell8/chunkmaster")

if (TryAcquire(x) == success) {
   // I'm the chunkmaster, tell everyone
   SetContents(x, my-address)

} else {
   // I'm not the master, find out who is
   chunkmaster = GetContents(x)
}
```

Consider that **x** is a variable (or file) with the "leader id" information

The path is the address of **x** in data model

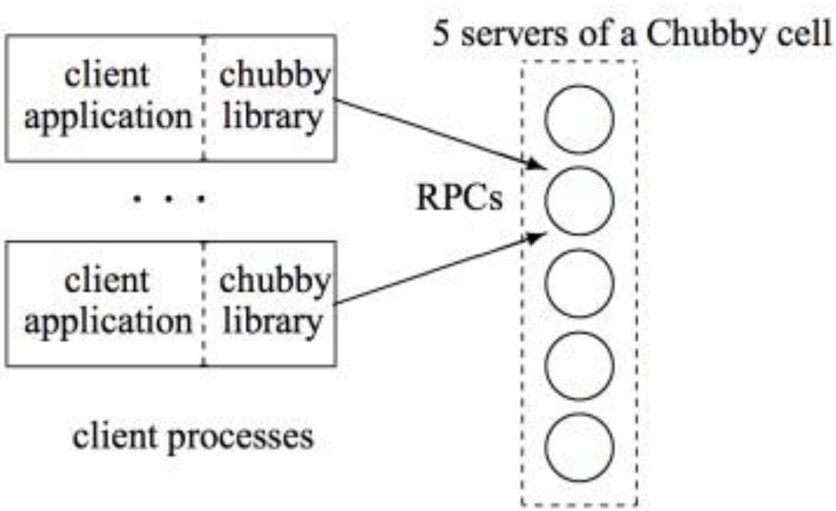
- It is a **standard repository for files that require high availability**
  - E.g. access control lists
  - E.g. configuration files

Chubby Design

- Chubby has two main components that communicate via RPC: a **server**, and a **library** that client applications link against
  - All communication between Chubby clients and the servers is mediated by the client library

- Chubby achieves **fault-tolerance** through **replication**

- A **Chubby cell** consists of a small set of **servers** (typically **five**) known as **replicas**
  - Replicas are placed so as to reduce the likelihood of correlated failure
    - For example, in different racks
  - Replicas use a distributed consensus protocol to elect a **master**

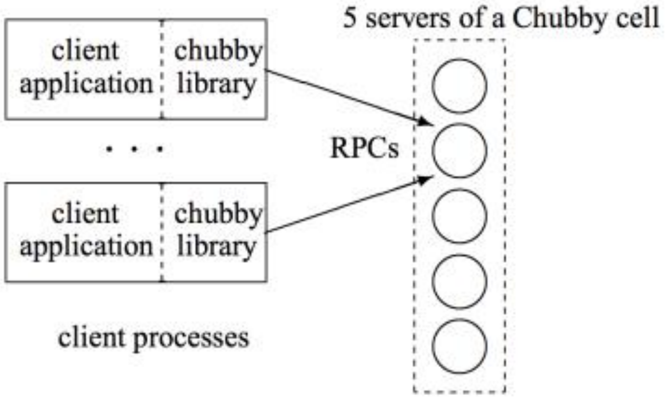


Chubby Design

- The **replicas** maintain **copies of a simple database**
  - Only the **master initiates reads and writes** of this database
  - All other **replicas simply copy updates from the master**, sent using the consensus protocol

- Detail…
  - Clients find the master by sending master location requests to the replicas listed in the DNS
  - Non-master replicas respond to such requests by returning the identity of the master
  - Once a client has located the master, the client directs all requests to it either until it ceases to respond, or until it indicates that it is no longer the master
  - **Write** requests
    - They are propagated via the consensus protocol to all replicas
    - Such requests are acknowledged when the write has reached a majority of the replicas in the cell
  - **Read** requests
    - They are satisfied by the master alone
    - This is safe since the "master lease" has not expired, as no other master can possibly exist

The period of time in which the elected master is the leader



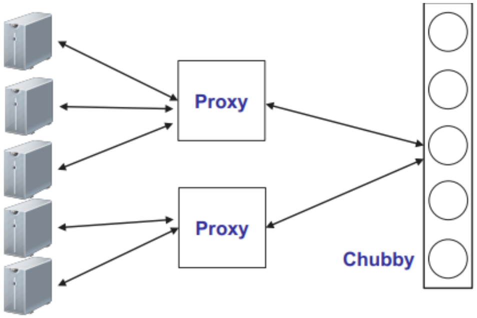
Chubby Design

- **Master** is performance **bottleneck**
- If master fails…
  - Another replica must propose itself as master
  - Other replicas run the **election protocol** when their "master leases" expire
- If replica fails…
  - A **replacement system** selects a fresh machine from a free pool and starts the lock server binary on it
    - It then updates the DNS tables, replacing the IP address of the failed replica with that of the new one
  - The current master polls the DNS periodically and eventually notices the change
    - Master then **updates the list of the cell's members in the cell's database**
    - This list is kept consistent across all the members via the normal replication protocol
  - In the meantime, the **new replica** obtains a recent copy of the database from a combination of **backups** stored on file servers and **updates** from active replicas
  - Once the new replica has processed a request that the current master is waiting to commit, the replica is permitted to vote in the elections for new master

Scaling Chubby

Chubby's normal workload:
- ~93% KeepAlive
- ~1% Read
- <1% Write

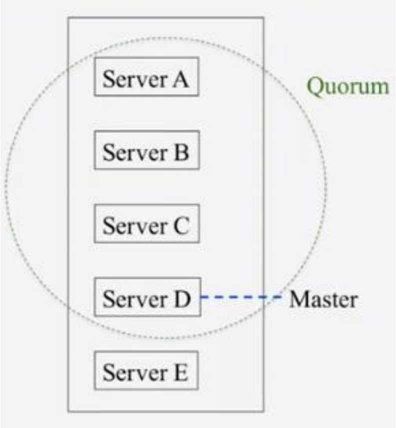
- Using **cache** in clients
  - Clients cache data they read
  - Master invalidates cached copies upon update
  - Master must store knowledge of client caches
- Using **proxies**
  - Chubby's protocol can be proxied by trusted processes that pass requests from other clients to a Chubby cell
  - A **proxy can reduce server load** by handling both KeepAlive and read requests
    - A Chubby session is a relationship between a Chubby cell and a Chubby client; it exists for some interval of time, and is maintained by periodic handshakes called **KeepAlives**
  - It cannot reduce write traffic, which passes through the proxy's cache



- [Burrows, 2006] It scaled to **tens of thousands of client processes** per Chubby instance

Election in Chubby

- One possible consensus approach to solve election
  - Each process P_i 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 practice
  - Use Paxos-like approaches for election
    - Lamport's Paxos is a family of protocols for solving consensus in a network of unreliable processors
- 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, and informs everyone
  - Election time [Burrows, 2006]:
    - In general: 4s or 6s
    - Worst case: 30s



- Safety
  - Each potential leader try to reach a *quorum*
  - Since any two *quorums* intersect and each server votes at most once, we cannot have two leaders elected simultaneously
- Liveness
  - It is not guarantee
  - Failures may keep happening so no leader is elected
- There is a "promise" that servers will not elect a different master for an interval of a few seconds known as the "**master lease**"
  - Lease technique ensures automatic re-election on master failure

Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

ZooKeeper

- Apache License
  - Written in Java
  - zookeeper.apache.org
  - First version in **2008**

- A distributed **open-source** centralized **coordination service**

  - It is a service used by a **cluster** (group of nodes) to **coordinate** between themselves and **maintain shared data** with robust **synchronization** techniques

  - It is itself a distributed application providing services for writing a distributed application

ZooKeeper Typical Use Cases

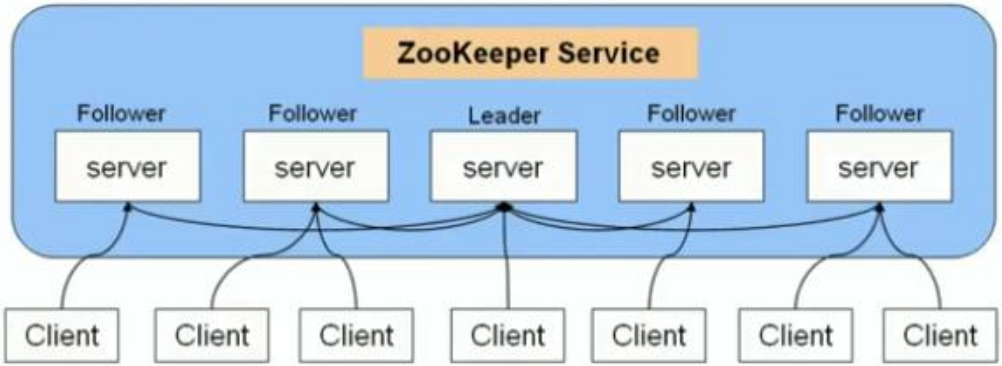
- Naming service
  - Identifying the nodes in a cluster by name. It is similar to DNS, but for nodes
    - A naming service is a service that maps a name to some information associated with that name
    - In your distributed system, you may want to **keep a track of which servers or services are up and running and look up their status by name**
- Configuration management
  - Latest and up-to-date configuration information of the system for a **joining node**
    - The **configuration of your distributed system must centrally stored and managed**
    - This means that any new nodes joining should pick up the up-to-date centralized configuration as soon as they join the system
- Leader election
  - Electing a node as leader for coordination purpose
    - Your distributed system may have to deal with the **problem of nodes going down**, and you may want to implement an **automatic fail-over strategy**. You can do this by leader election.
- Locking and synchronization
  - Locking the data while modifying it
    - To allow for serialized **access to a shared resource** in your distributed system, you may need to implement distributed mutexes
- Highly reliable data registry
  - Availability of data even when one or a few nodes are down

Projects that use ZooKeeper

- Apache BookKeeper
  - BookKeeper (ZooKeeper subproject) is a replicated service to reliably **log** streams of records
- Apache Hadoop MapReduce
  - The next generation of Hadoop MapReduce (called "Yarn") uses ZooKeeper
- Apache Hbase (Hadoop database)
  - HBase uses ZooKeeper for **master election**, **server lease management**, **bootstrapping**, and **coordination between servers**
- Apache Kafka
  - Kafka is a distributed publish/subscribe messaging system.
  - Kafka queue consumers uses Zookeeper to **store information** on what has been consumed from the queue
- Apache Storm
  - Storm uses Zookeeper to store **all state** so that it can **recover** from an outage in any of its (distributed) component services.
- Others cases: Rackspace, Twitter, Vast.com, Yahoo, Zynga…

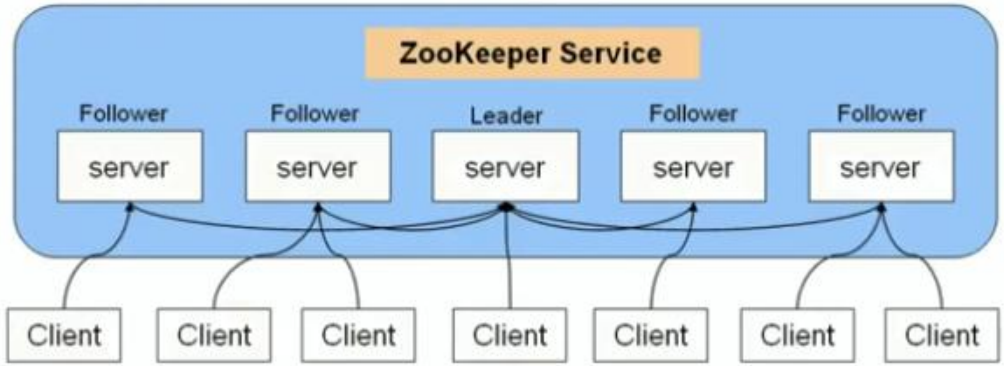
ZooKeeper Design

- Zookeeper service runs on a **cluster** of machines (or **ensemble**)
  - Cluster has **cluster nodes**
- Server Applications
  - The server is distributed and has a centralized interface through which the clients can connect to the service
  - One is the **leader** and the other are **followers**
  - If **leader** fails, we need to **elect** other
  - Servers know about each other
- Client Applications
  - They are basically the tools that are available for interacting with the Zookeeper distributed application
  - These clients could be command line or GUI client
  - **Clients connect to only one server**
    - If they lose connection, can connect to other server automatically

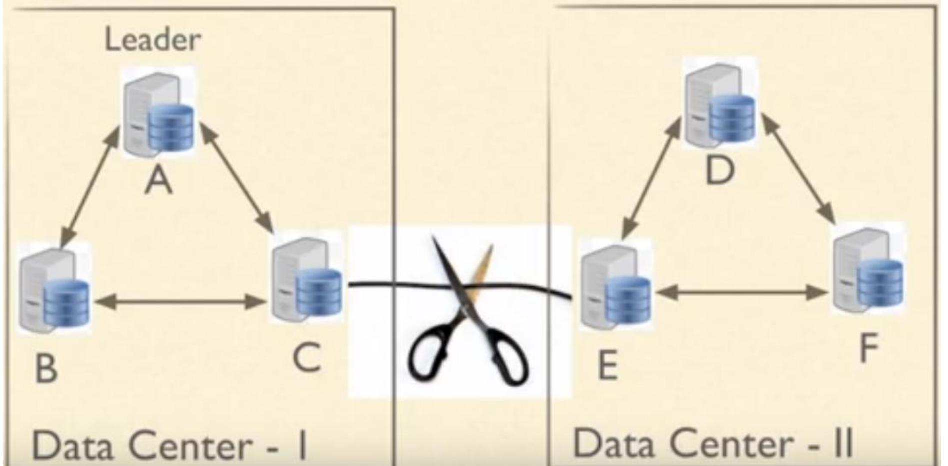


ZooKeeper Design

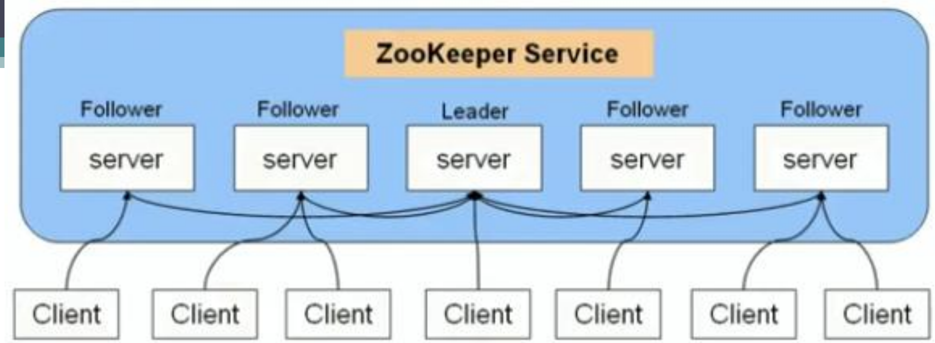
- Zookeeper service
  - Provides high availability and consistency
    - Requires the **majority of servers** (to be up and running at all times)
    - E.g. 7 node ensemble can lose 3 nodes



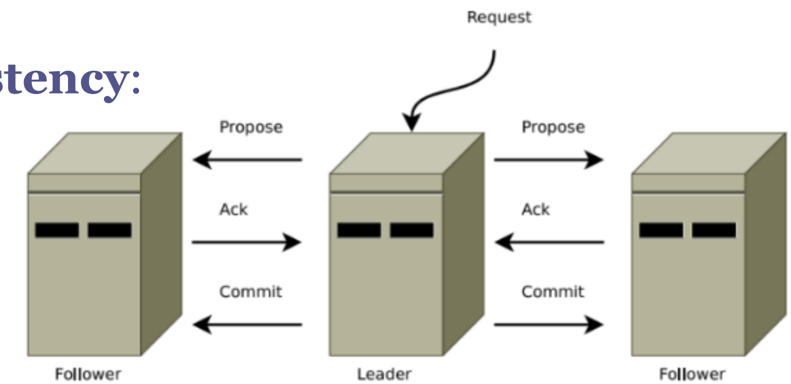
- But why we need **majority**?
  - E.g. Each datacenter (now isolated) could elect a leader, driven to inconsistency!



ZooKeeper Design



- Read operation
  - **In any server**
  - Reads are **concurrent** since they are served by the **specific server that the client** connects to
    - It is the reason for the **eventual consistency**:
      - The "view" of a client may be outdated, since the master updates the corresponding server with a bounded but undefined delay



- Write operation
  - The **master is the authority for writes**
  - All requests that **update ZooKeeper state (writes)** are forwarded to the **leader**
    - The leader executes the request and broadcasts the change to the ZooKeeper state using **ZAB (Zookeeper Atomic Protocol)**
    - ZAB is a variant of Paxos
    - ZAB uses by default simple majority **quorums** to decide on a proposal
      - The quorum include the server for the client, and obviously the leader
      - This means that each write makes the given server up-to-date with the master
      - It also means, however, that you cannot have concurrent writes

ZooKeeper Consistency Guarantees

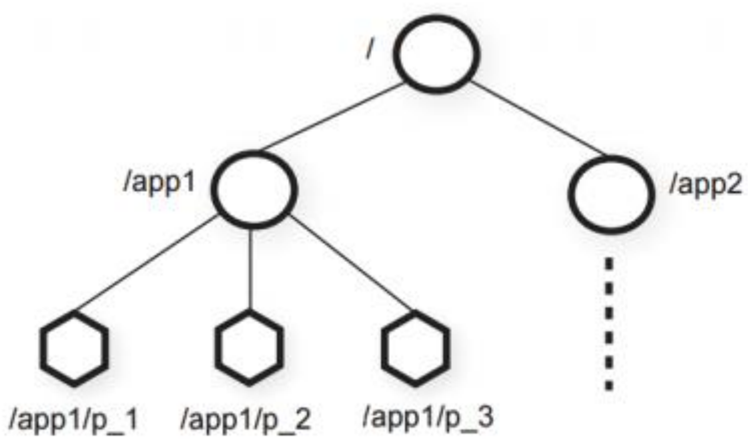
- Sequential consistency
  - Clients updates are applied in order
- Atomicity
  - Updates succeed OR fail
  - Partial updates are not allowed
- Single system image
  - Client will see the same view of ZooKeeper service regardless of server
- Reliability
  - If update succeeds then it persists
- Timeliness
  - Client view of system is guaranteed up-to-date within a time bound
    - Generally with 10 seconds
    - If client does not see system changes within time bound, then service-outage (so client connects to other server)

- Note: ZooKeeper does not guarantee simultaneously consistent cross-client views
  - Different clients will not always have identical views of ZooKeeper data at every instance of time
  - We can "force" a follower server to "catch up" with leader, using *sync()* method

ZooKeeper Data Model

- The **replicated database** of ZooKeeper comprises a **tree of znodes**
  - Distributed processes then coordinate through this shared hierarchical namespaces
  - The tree of **znodes** is similar to tree of files/directories

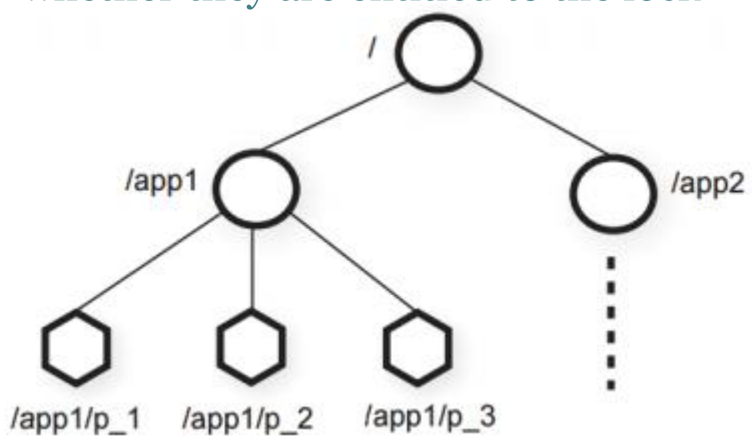
- znode
  - It can be updated and/or modified by any node in the cluster
  - It holds data, children or both
  - It has an ACL (Access Control list)
    - To inform who can create, read and update a znode
  - It can be sequential and ephemeral

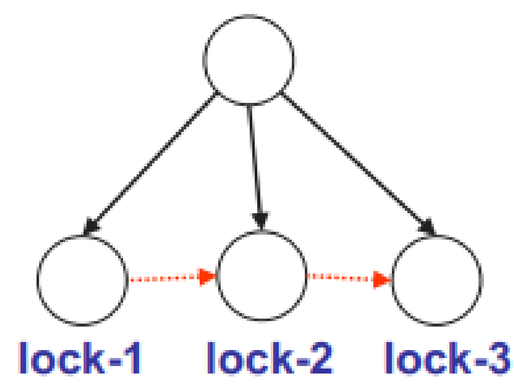


ZooKeeper Data Model

- Sequential znode
  - The name the **client** provides when creating the znode is only a **prefix**
  - The **full** name is also given by a **sequential number** chosen by the ensemble
  - It is useful, for example, for **synchronization** purposes
    - E.g. If multiple clients want to get a lock on a resource, they can each concurrently create a sequential znode on a location: whoever gets the **lowest number is entitled to the lock**

- Ephemeral znode
  - It is **destroyed** as soon as the client (that created it) disconnects
  - They can not have children
  - This is mainly useful in order **to know when a client fails**, which may be relevant when the client itself has responsibilities that should be taken by a new client
  - E.g. As soon as the client having the lock disconnects, the other clients can check whether they are entitled to the lock





ZooKeeper Data Model

- **Watches**
  - The example related to **client disconnection** may be **problematic if we needed to periodically poll** the state of znodes
  - Fortunately, ZooKeeper offers an event system where a *watch* can be set on a znode.
    - These **watches may be set to trigger an event** if the znode is specifically changed or removed or new children are created under it

znode with my lock

All locks for that resource

Lock

```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

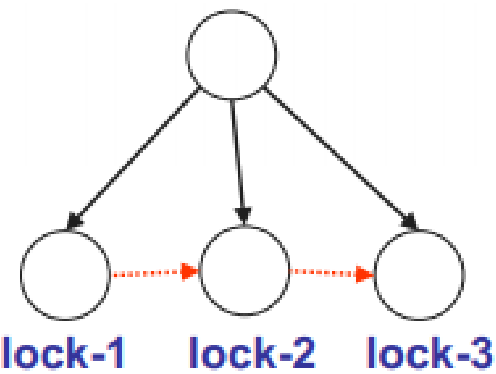
I got the lock!

p has priority for lock

If p did not fail, wait it exits CS.
Later I will try again to have the lock

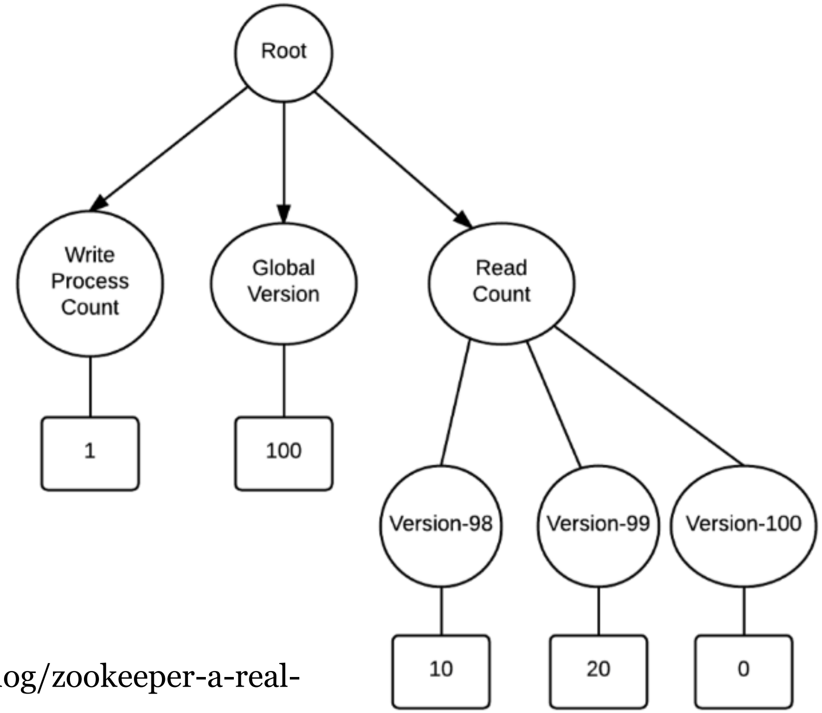
Unlock

```
1 delete(n)
```



ZooKeeper Real Use Case

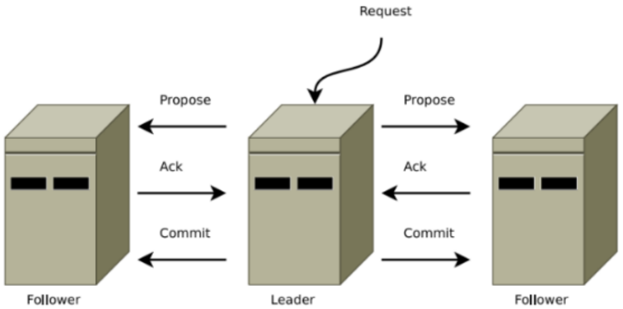
- A multi-version system
  - The “global version” is 100
  - There are 10 and 20 read requests being executed on versions 98 and 99 respectively
  - Since there is a write request in progress, no other write request would be taken up until it completes
  - Once Reads in Version-98 finish, we can archive this version



Reference: https://resources.zaloni.com/blog/zookeeper-a-real-world-example-of-how-to-use-it

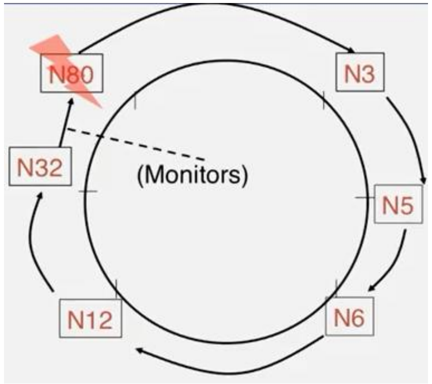
Election in ZooKeeper

- Election protocol
  - Each server creates a new sequence number (**id**) for itself
    - By getting the highest id (from ZK file system), creating other and storing in ZF file system again
  - Elect the **highest id** server as **leader**
    - There will be only one id in ZK file system
- Election uses ZAB (ZooKeeper Atomic Broadcast)
  - Again… ZAB is a variant of Paxos
  - Leader sends NEW_LEADER message to all
  - Each server responds ACK to at most one leader (with highest id)
  - Leader waits for a majority (*quorum*) of ACKs, and then send COMMIT
  - On receiving COMMIT, process updates its leader variable



- If master fails…
  - One option:
    - Everyone monitors the current master
    - One failure, initiate election
  - ZK option:
    - Each process monitors the next higher-id server
    - **IF** that successor is leader and fails
      - I become the new leader!
      - We do not need to run a new election protocol
    - **ELSE**
      - Wait for a timeout, and check the successor again
      - It is possible to run a new election protocol…

Leads to a flood of elections ⇨ Too many messages

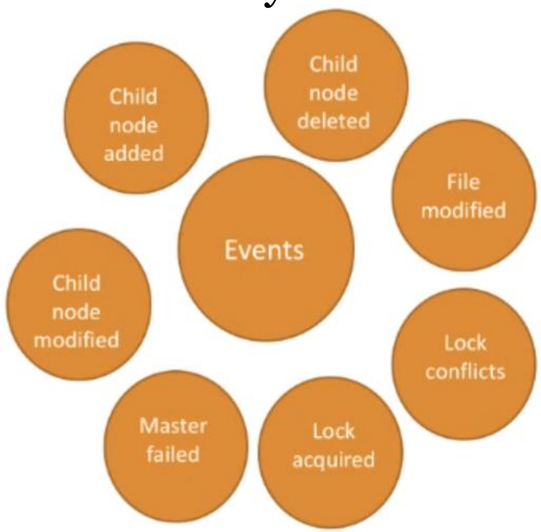


- Safety
  - Guarantee by the use of *quorums*
- Liveness
  - It is not guarantee.
  - We may need to run the election again

Chubby and ZooKeeper

- Similarities
  - They can be used as **distributed lock service**
  - They read/write **small files**
  - They provide a mechanism to follow changes on files (**events and watches**)
  - Clients have to go over **leader/master for write** operations

Chubby events



Zookeeper watches

- Main difference
  - Chubby: Reads through master
  - ZK: Reads through any server

Outline

- Mutual exclusion
- Token-based algorithms
  - Centralized algorithm
  - Token ring algorithm
- Timestamp-based algorithms
  - Ricart-Agrawala
- Quorum-based algorithms
  - Maekawa
- Cases
  - Chubby
  - ZooKeeper

Extra slides

Token ring algorithm

- Token loss detection and regeneration
  - **Timeout-based algorithm**
    - Processes are numbered in a monotonically growing sequence
    - Set the value v of the token between 0 and k-1 (k>2)
      - Each site Pi keeps state information recording the token value: VAR state: 0..k-1
    - Before token transmission
      - Pi must: IF i=0 THEN v := v+1 mod k; //only P0 updates token
        state:= v;
      - Note that the value of v is the same for each complete cycle around the ring
    - Detection of token loss by Pi:
      - Every time a token is transmitted, Pi starts a timer
      - If timeout period expires before return of the token
        - Then ask the previous Pj, where j = neighbor[left, i].
        - The token is considered lost if Pj already had the token, but it was not received by Pi => check state variable!

Token ring algorithm

- Token loss detection and regeneration
  - **Timeout-based algorithm (cont)**
    - Token regeneration by Pi:
      - Regenerate the token with the correct value v (state of Pj)
    - Disadvantages:
      - Although simple, the algorithm depends on **timeouts** to allow all sites use the token for mutual exclusion
        - i.e. timeout > sum (exclusion time + protocol time)
        - What happens if one process begins a token detection and it is not lost yet?
      - Processes identities needed
        - One process must be the coordinator (P0)
        - Symmetry is needed (process ids)

Token ring algorithm

- Token loss detection and regeneration
  - **Misra algorithm (Ping-pong algorithm)**
    - Reference: Misra, Jayadev. "Detecting termination of distributed computations using markers." Proceedings of the second annual ACM symposium on Principles of distributed computing. ACM, 1983.
    - Neither requires knowledge of **delays** nor process **identities**
    - Two tokens: '**ping**' and '**pong**'
      - The tokens circulate in the same direction
        - Suppose the preservation of messages order in a connection
        - Assume that the tokens don't overtake each other
      - Only one token is associated with obtaining the exclusion
    - Each token is used to detect the possible loss of the other:
      - Loss detected in Pi if in a complete passage around the ring neither Pi, nor the token (not lost) found other token
      - Token associated values: **nbping** and **nbpong**
      - They have equal absolute values but opposite in sign
        - In order to record the number of times the tokens have met
      - Each time they meet, nbping and nbpong are incremented in module: nbping++ and nbpong--
      - Invariant : nbping + nbpong = 0
      - Initially, nbping = 1; nbpong = -1

Token ring algorithm

- Token loss detection and regeneration
  - **Misra algorithm (Ping-pong algorithm) (cont)**

Interface that only receives message (token) of type integer

```
PROCESS P (i: integer):
        PORT ping-pong : integer;
        PORTREF outping-pong : integer;
        VAR
            nbping, nbpong : integer;
            token, t: integer;
            pinghere, ponghere : Boolean;
```

Interface to send message of type integer

Last token value

New token value

```
BEGIN
token:=0;
...
ping-pong.IN(t)
=>  IF t>0 THEN                          {PING ou PONG?}
        BEGIN                            {PING!}
        pinghere:=true; nbping:=t;
        IF ponghere THEN                 {tokens se encontram?}
            BEGIN
            nbping:=nbping+1;
            nbpong:=nbpong-1;
            END
        ELSE IF token <> nbping THEN     {PONG foi encontrado?}
            token := nbping
        ELSE
            BEGIN { ping deu uma volta e não encontrou pong}
            nbping:=nbping+1;
            nbpong:=-nbping
            END
        END
    ELSE                                 {PONG!}
    { como acima com pings and pongs trocados }
    END
...
token:=nbping;   (ou nbpong como apropriado)
pinghere:=false;   (ou ponghere:= false)
outping-pong.SEND(nbping);   (ou nbpong)
END.
```