CLOUD COMPUTING

UNIT-4

Cloud Controller

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

CLOUD CONTROLLER

CONTEXT of CLOUD CONTROLLER- EUCALYPTUS

- Started as research project in UC Santa Barbara
- Paid and open-source software for building AWS-compatible private and hybrid cloud computing environments
- Iaas - hybrid cloud - easy migration of data between public AWS cloud and on-premises private Eucalyptus cloud (hybrid cloud deployment)

https://www.sciencedirect.com/topics/computer-science/cloud-controller

Terminology

- UFS (User-Facing Services) – implements web service interfaces that handle AWS-compatiple APIs
  - Requests from CLI or GUI

- CLC (Cloud controller) – high-level resource tracking, management, resource allocation, task scheduling, accounting
  - Only one CLC per Eucalyptus cloud

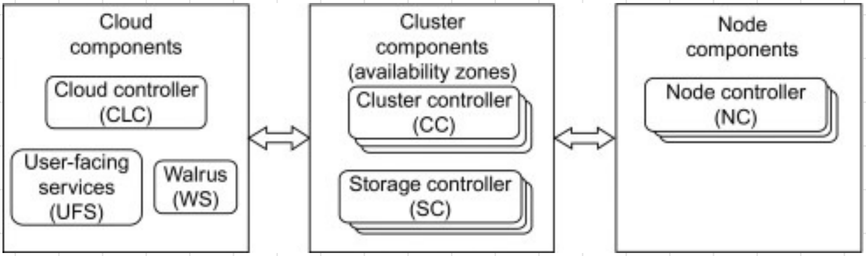
- CC (Cluster Controller) – manages and deploys VM instances, manages node controllers and storage controllers

- NC (Node Controller) – runs on the machines that hosts VMs and manages endpoints
  - No limit to number of NCs
  - Interacts with OS and hypervisor to maintain lifecycle of instances

Storage services

- SC (Storage Controller) – similar to AWS EBS (elastic block storage)
  - Provides block-level storage for VM instances

- WS (Walrus) – similar to AWS S3 (persistent storage)



R3

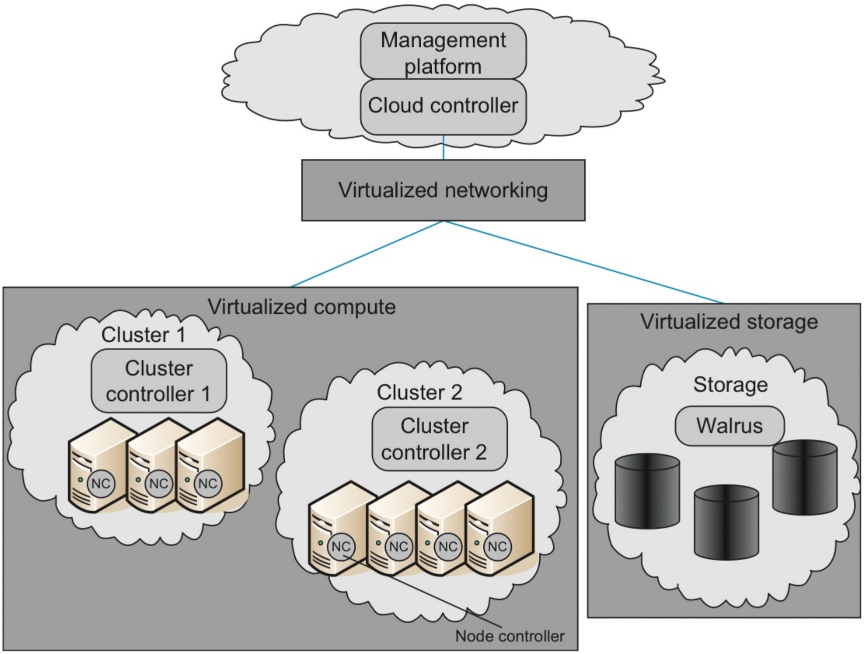
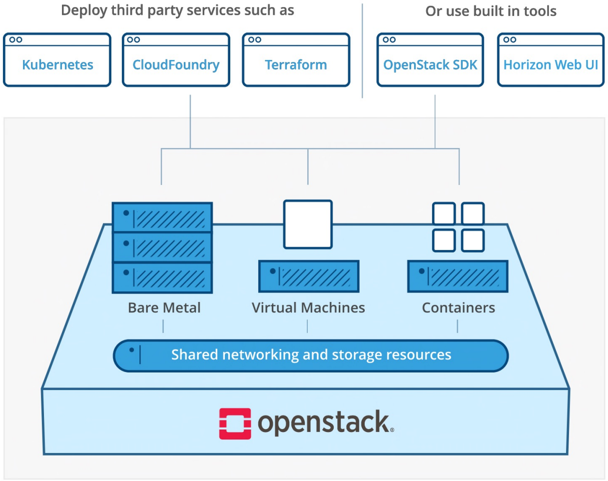


FIGURE 6.5

A schematic of key Eucalyptus modules.

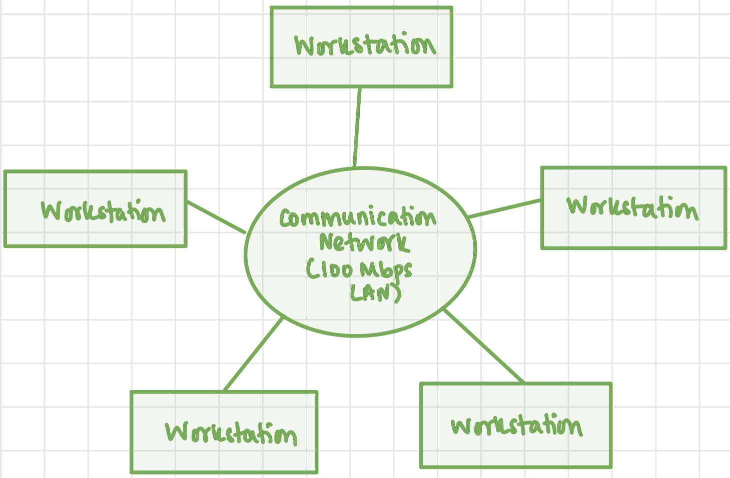
CONTEXT of CLOUD CONTROLLER- OpenStack



- Open source platform to manage public and private clouds
- Consistent APIs to abstract virtual resources

DISTRIBUTED SYSTEMS

- System with multiple components located on different machines that communicate in order to appear as a single coherent system



Classification of Distributed System Models

1. Architectural Models

1.1 System Architecture

- How components of distributed system are placed across multiple machines
- How responsibilities are distributed
- Eg: P2P, client-server

1.2 Software Architecture

- Logical organization of software components
- Eg: 3 Tier Architecture

2. Interaction Models

- Handling of time
- Limits on process execution, message delivery, clock drifts
- Eg: Synchronous DS, Asynchronous DS

3. Fault Models

- Types of faults that can occur and their effects
- Eg: Omission faults, Arbitrary faults, Timing Faults

System Architecture Models

1. Peer-to-Peer Systems

- Every node acts as both client and server
- Peers autonomous in joining and leaving the network
- Self-organizing
- Peers form a virtual overlay network on top of the physical network topology

2. Client-Server Systems

- Client asks server for a service
- One server can serve several clients
- One client can request services from several servers
- Master-slave with single point of failure

Building Reliable Distributed Systems with Unreliable Components

- Underlying communication network unreliable, commodity hardware prone to failure
- Cloud resources need to constantly monitor infrastructure to ensure reliability
- Individual computer: either fully functional or entirely broken (hardware problems, memory corruption etc.) — no in between (deterministic)
- Distributed systems: partial failure possible and common (non-deterministic)
- Internal networks in datacentres are asynchronous packet networks (no guarantee on packet delivery time)
- If you send a request and expect a response — things that could go wrong

1. Your request may have been lost (perhaps someone unplugged a network cable).
2. Your request may be waiting in a queue and will be delivered later (perhaps the network or the recipient is overloaded).
3. The remote node may have failed (perhaps it crashed or it was powered down).
4. The remote node may have temporarily stopped responding (perhaps it is experiencing a long garbage collection pause; see "Process Pauses" on page 295), but it will start responding again later.
5. The remote node may have processed your request, but the response has been lost on the network (perhaps a network switch has been misconfigured).
6. The remote node may have processed your request, but the response has been delayed and will be delivered later (perhaps the network or your own machine is overloaded).

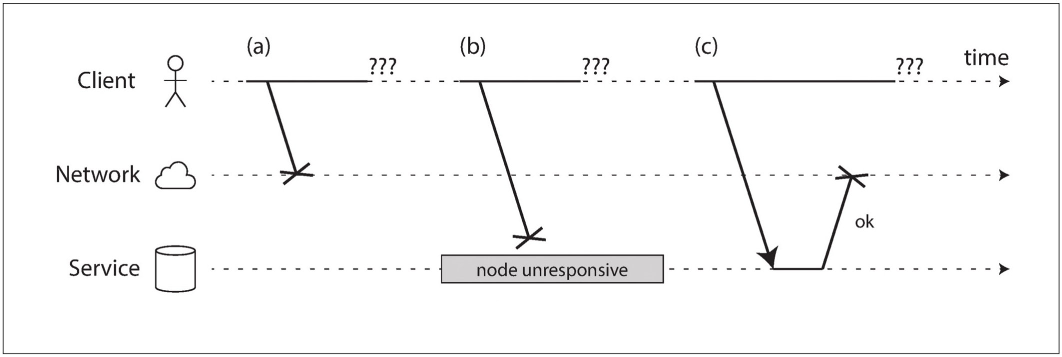


Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

- Reliability: probability that system meets performance standards and yield correct output in a specified time period

Summarizing Failures and Faults

- System or component fails when it cannot meet its promises
- Failures due to errors caused by faults

Types of Faults

1. Transient Faults:
   - Appears once and then disappears
2. Intermittent Faults:
   - No pattern of occurring, vanishing, reappearing
3. Permanent Faults:
   - Component needs replacement

Mean Time Between Failures

MTBF = average length of operating time b/w failures

$$MTBF = \frac{\text{total uptime}}{\#\text{ of failures}}$$

Fault Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition f.*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary (Byzantine) failure	A server may produce arbitrary responses at arbitrary times

Issues that Lead to Faults in Distributed Systems

1. Timeouts and Unbounded delays
   1.1 Network Congestion and queueing

1. Timeouts and Unbounded Delays

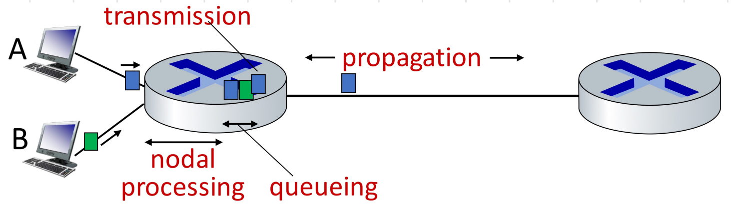
- Ideal system: guarantee on packet delivery time d — every packet is either delivered within time d or is lost — and request handling time r
  - If true, 2d+r is reasonable timeout

- Real system: asynchronous networks with unbounded delays
- Experimentally choose either constant timeout or dynamic timeouts adjusted based on network response times

1.1 Network Congestion and Queueing

- Variability of packet delay in networks is due to queueing delays

http://gaia.cs.umass.edu/kurose_ross



$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

$d_{prop} = \frac{d}{s}$ ← d: length of physical link; s: prop speed ($\sim 2\times10^8 ms^{-1}$)

d_{queue} = time waiting in buffer (o/p link) for trans, μs to ms ← congestion

$d_{trans} = \frac{L}{R}$ ← L: no. of bits/packet; R: transmission rate

d_{proc} = nodal processing, determine output link, typically < ms (from header)

- If several nodes (port 1,2,4) try to send packets to same destination, congestion on outgoing link
- If packet lost, TCP retransmits (added delay)

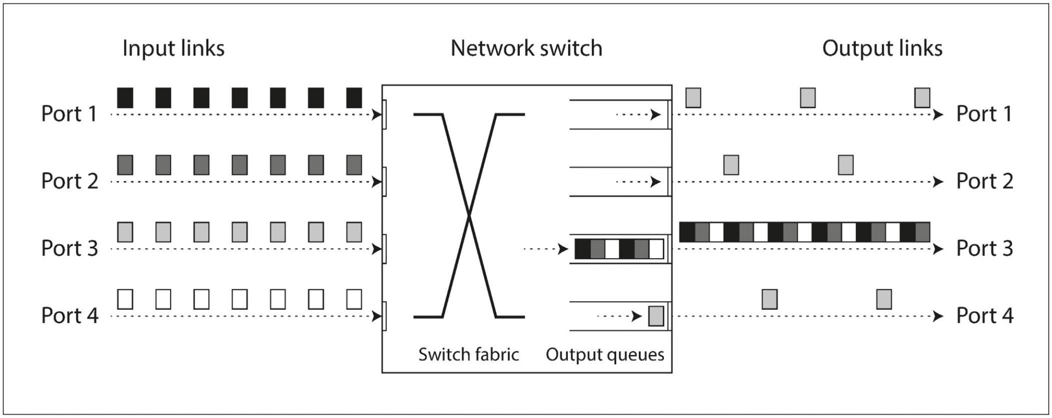


Figure 8-2. If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2, and 4 are all trying to send packets to port 3.

- If destination system's CPU cores are all occupied, incoming network request queued by OS

- Virtualized environments: different VMs running on hardware context switch on cores, pausing execution ; VMM queues incoming packets

- Queueing at sender end (TCP flow control)

- In public clouds / multi-tenant datacenters, all resources are shared
  - Noisy neighbour can hog resources
  - Batch workloads like Map-Reduce can use up links

OMISSION & ARBITRARY FAULTS

1. Crash-stop faults
   - Node suddenly stops responding (permanent)
   - May or may not be detectable

2. Crash-recovery faults
   - Nodes crash and may restart after an unknown period of time
   - Assumptions: storage safe but in-memory state lost

3. Byzantine (arbitrary) faults
   - Node can do anything unpredictably

Detection of Faults

1. Heartbeats
   - Each app periodically sends signal
   - If heartbeat not sent for a specified period, failure

2. Probing
   - Monitoring service periodically sends probes (lightweight service requests) to app instance
   - Decide based on response

STRATEGIES for DEALING with PARTIAL FAILURES

1. Asynchronous communication across internal microservices
   - Eventual consistency
   - Event-driven architecture

2. Retries with Exponential Backoff

3. Work around network timeouts
- Clients should not block
- Timeouts for responses

4. Use circuit breaker pattern
- Client process tracks no. of failed attempts
- If no. of failed attempts > threshold within a period of time, 'circuit breaker' trips
- All subsequent requests immediately fail until timeout period ends
- After timeout, request sent again

5. Provide fallbacks
- If request fails, client itself performs fallback (return cached/ default data)

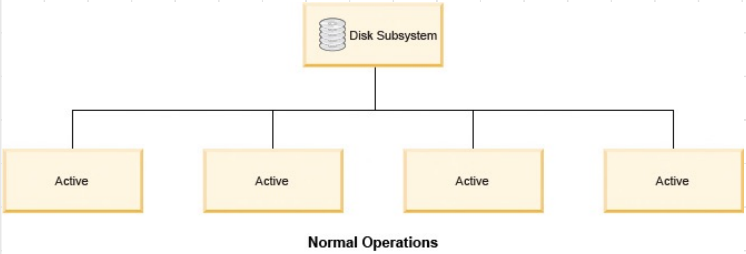
6. Limit no. of queued requests
- Limit on outstanding requests that client microservice sends to particular service
- Polly Bulkhead Isolation Policy

Failover Strategy

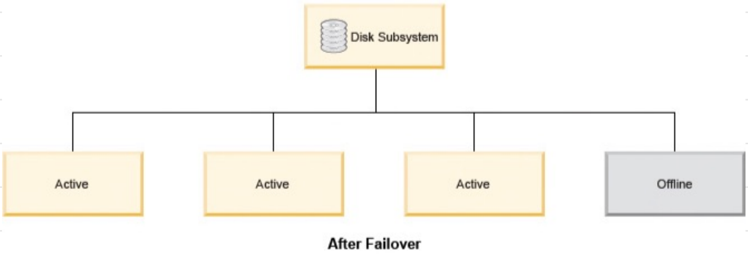
- Failover: switch to replica upon failure of previously active application
- Strategies
  1. Active-Active / Symmetric
  2. Active-Passive / Asymmetric

1. Active-Active

- Configuration typically used for load-balancing
- 2 or more nodes run same application/service using same database server
- All nodes actively processing transactions



- Event of failure, other nodes handle load and continue to provide service



- Continuous availability

2. Active-Passive

- Fully redundant instance of each node
- Only brought online if primary fails

SYSTEM AVAILABILITY

- Period for which a service is available and works as required (percentage)

- Terminology

  1. Uptime : time for which system is running
     - Typically percentage (99.999% or 5 9's)

  2. Downtime : outage duration

- Service-level agreement contracts typically include uptime assurance

Q: A website was monitored for 24 hours. The monitor was down for 10 minutes. What were the uptime % and downtime % ?

Total time = 24 hours = $24 \times 60 = 1440$ mins
Total downtime = 10 mins

$$\therefore \text{uptime } \% = \frac{1440 - 10}{1440} \approx 99.305\%$$

$$\text{downtime } \% = \frac{10}{1440} \approx 0.695\%$$

High Availability

- Design distributed system environment such that
  1. All single points of failures removed through redundancy
  2. Faults tolerated through automatic failover to backups

- Virtually no downtime

FAULT TOLERANCE

- System's ability to continue operating uninterrupted despite failure of one or more components
- Types of fault tolerance
  1. Fail-safe fault tolerance
  2. Graceful degradation

Considerations for Building Fault Tolerance

Some important considerations when creating fault tolerant and high availability systems in an organizational setting include:

- **Downtime** – A highly available system has a minimal allowed level of service interruption. For example, a system with "five nines" availability is down for approximately 5 minutes per year. A fault-tolerant system is expected to work continuously with no acceptable service interruption.
- **Scope** – High availability builds on a shared set of resources that are used jointly to manage failures and minimize downtime. Fault tolerance relies on power supply backups, as well as hardware or software that can detect failures and instantly switch to redundant components.
- **Cost** – A fault tolerant system can be costly, as it requires the continuous operation and maintenance of additional, redundant components. High availability typically comes as part of an overall package through a service provider (e.g., load balancer provider).

Approaches for Building Fault Tolerance

1. Redundancy: avoid single points of failure with hardware and software redundancies
2. Reliability: dependability; analyzed based on failure logs, frequency

(a) Mean Time Between Failures

- Average time between repairable failures
- Higher better

$$MTBF = \frac{\text{total operational time}}{\text{\# of failures}}$$

(b) Mean Time to Failure

- Average time between non-repairable failures

$$MTTF = \frac{\text{total uptime of all systems}}{\text{\# of systems (that failed)}}$$

Q: There are 3 identical systems that start at time $t = 0$. All 3 of them eventually fail. Uptimes: 10 hours, 12 hours, 14 hours. Find MTTF.

$$MTTF = \frac{10 + 12 + 14}{3} = 12 \text{ hours}$$

3. **Repairability**: how quickly failing parts can be repaired.

(a) Mean Time to Recovery

- Time taken to repair system

$$MTTR = \frac{\text{total downtime of system}}{\text{\# of failures}}$$

Q: A system fails 3 times a month and results in 6 hours of downtime. Find MTTR.

$$MTTR = \frac{6}{3} = 2 \text{ hours}$$

4. **Recoverability:** ability to overcome momentary failure so no impact on end-user availabity

Additional Fault Tolerance Techniques

1. Retries
2. Timeouts
3. Circuit breakers
4. Isolate failures
5. Cache
6. Queue
7. Two phase commit

System Availability

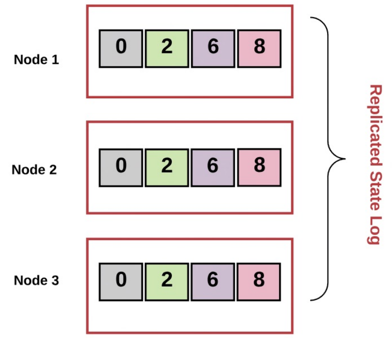
$$\text{System availability} = \frac{MTTF}{MTTF + MTTR}$$

CONSENSUS

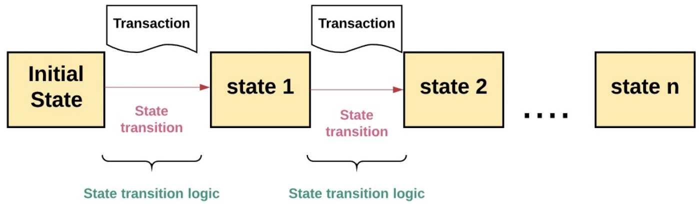
- Distributed database transactions — computes must collectively agree on the trancaction output
- Consistent trancaction logs
- Goal of concensus algorithm — all systems in the same state
- State trancition diagram

Replicated State Machine

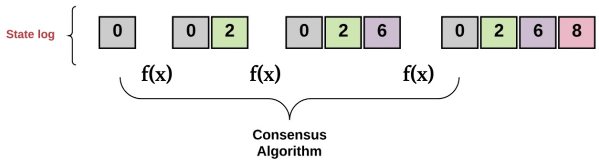
- Architecture to represent distributed systems
- Deterministic state machine replicated across multiple computers but functions as a single state machine



- If transaction is valid, input causes state to transition to next state according to state transition logic



- Systems must reach consensus to transition from one state to next state



CONSENSUS PROBLEM

- Consider a distributed system with N nodes
- An algorithm achieves consensus if it satisfies the following conditions
  1. Agreement: all non-faulty nodes decide on an identical output value
  2. Termination: all non-faulty nodes eventually decide on some output value
- Other basic constraints
  1. Validity
  2. Integrity
  3. Non-triviality
- Assumption: 3 types of actors in a system
  1. Proposers: leaders or coordinators
  2. Acceptors: listen to requests from proposers and respond
  3. Learners: other processes in the system that learn final values
- Generally, consensus algorithm defined by 3 steps

Step 1: Elect

- Processes elect a single process (i.e., a leader) to make decisions.
- The leader proposes the next valid output value.

Step 2: Vote

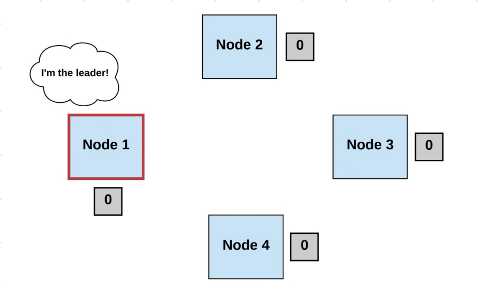
- The non-faulty processes listen to the value being proposed by the leader, validate it, and propose it as the next valid value.

Step 3: Decide

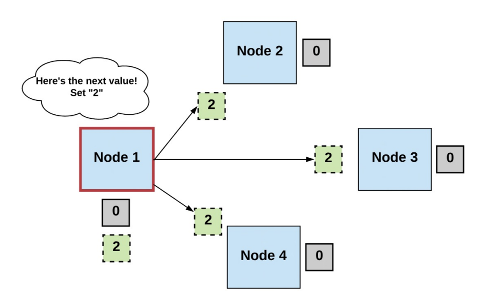
- The non-faulty processes must come to a consensus on a single correct output value. If it receives a threshold number of identical votes which satisfy some criteria, then the processes will decide on that value.
- Otherwise, the steps start over.

- Example

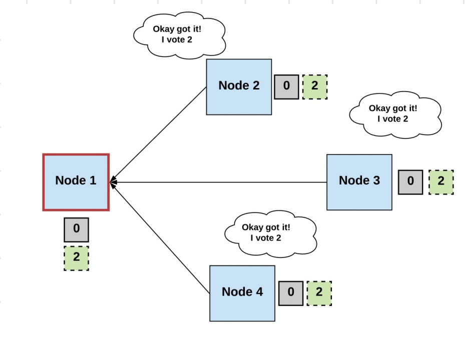
Step 1



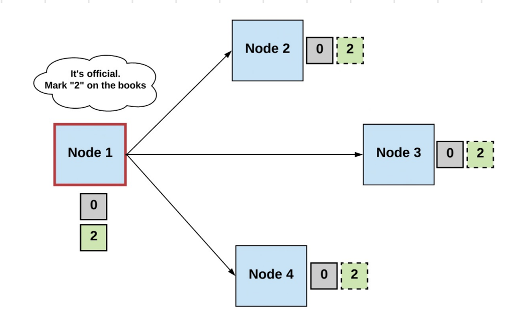
Step 2



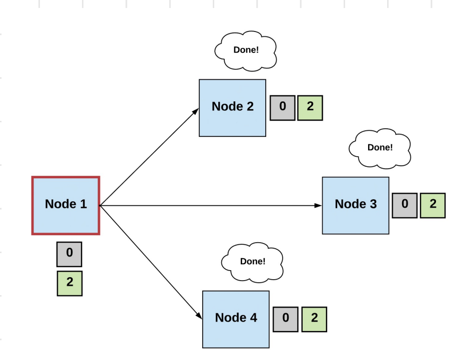
Step 3



Step 4



Step 5



Challenges in Arriving at Consensus

1. Reliable multicast
2. Membership failure detection
3. Leader election
4. Mutual election

Importance of Consensus Problem

- Many distributed systems problems are harder than or equivalent to the consensus problem
- If consensus problem can be solved, other problems can also be solved
- Problems equivalent or harder
  1. Failure detection
  2. Leader election

CONSENSUS in TWO SCENARIOS

1. Synchronous Distributed System
   - Can make assumptions about maximum message delivery time
   - Bound on local clock drifts
   - Consensus possible
   - Not practical to assume synchronous
2. Asynchronous Distributed System
   - No bound on process execution
   - Consensus impossible; there is always a worst-possible scenario
   - Probabilistic solutions only
   - FLP impossibility

Ways to Circumvent FLP Impossibility

- FLP impossibility:

> Even a single faulty process makes it impossible to reach consensus among deterministic asynchronous processes.

- Ways to tackle

crash fault tolerant ↗ Paxos, raft; Byzantine fault-tolerant ↗ DLS, PBFT

1. Use synchrony assumptions: Paxos, raft, DLS, PBFT
2. Use non-determinism: Nakamoto

PAXOS ALGORITHM

Phase 1: Prepare request

- Proposer: chooses new proposal version number n and sends "prepare request" to acceptors
- Acceptor: if received prepare request ("prepare", n, v) where $n >$ any other prepare requests previously responded to, acceptor sends out ("ack", n, n', v') where n' and v' are prev n and v
- Acceptor: promise not to accept proposals with number $< n$
- v: value of highest numbered accepted proposal (otherwise: v = ^, n = ^)

Phase 2: Accept Request

- Proposer: receives ack from majority of acceptors ⟶ issues accept request ("accept", n, v)

- n is same n from prepare request

- v is value of highest-numbered proposal among responses (v= max(sent v, received v's)

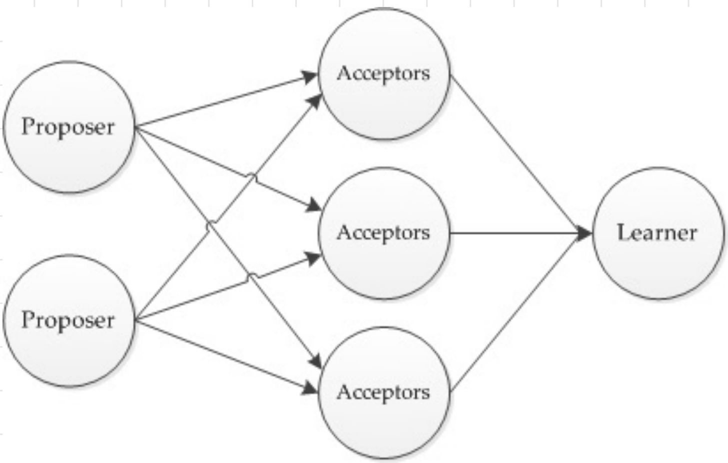
- Acceptor: if receives ("accept", n, v), accepts proposal unless it has already acked a prepare request with number > n

Phase 3: Learning Phase

- Acceptor: if accepts a proposal, responds to all learners with ("accept", n, v)

- Learners: receive ("accept", n, v) from majority of acceptors, send ("decide", v) to all other learners

- Learners: receive ("decide", v)



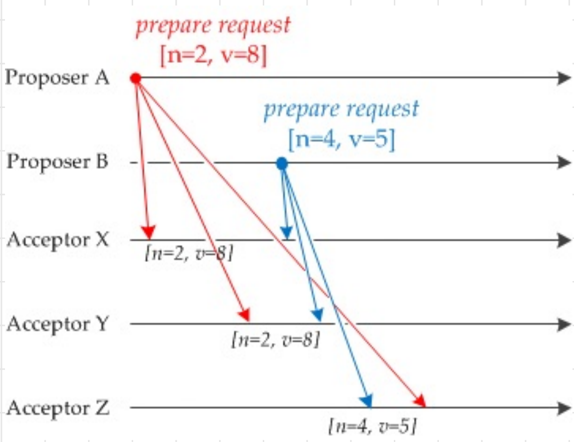
http://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf#page114

```
procedure Propose(n, v)
    // Issue proposal number n with value v
    // Assumes n is unique
    send prepare(n, v) to all accepters
    wait to receive ack(n, v′, n_v′) from a majority of accepters
    if some v′ is not ⊥ then
        v ← v′ with maximum n_v′
    send accept(n, v) to all accepters
procedure accepter()
    initially do
        n_a ← −∞
        v ← ⊥
        n_v ← −∞
    upon receiving prepare(n) from p do
        if n > max(n_a, n_v) then
            // Respond to proposal
            send ack(n, v, n_v) to p
            n_a ← n
    upon receiving accept(n, v′) do
        if n ≥ max(n_a, n_v) then
            // Accept proposal
            send accepted(n, v′) to all learners
            if n > n_v then
                // Update highest accepted proposal
                ⟨v, n_v⟩ ← ⟨v′, n⟩
```

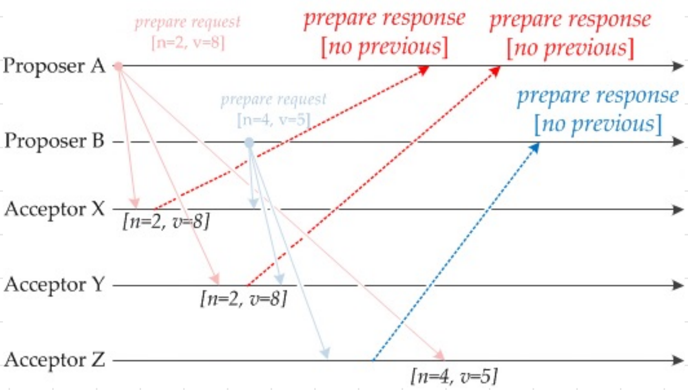
Algorithm 12.1: Paxos

Example: Proposers A, B and Acceptors X, Y, Z

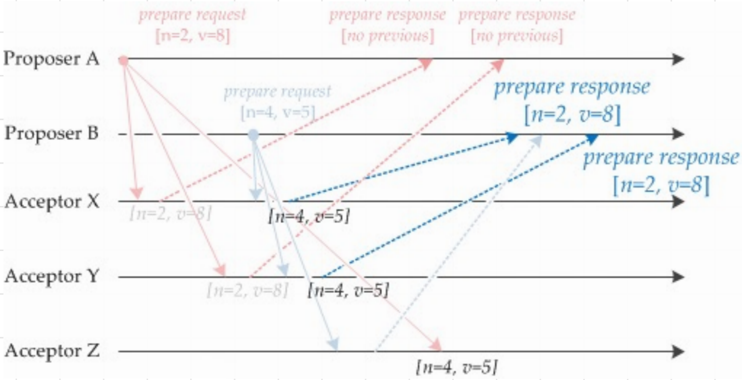
(1)



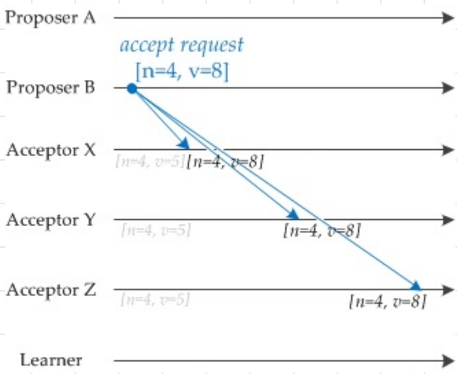
② No previous proposals accepted



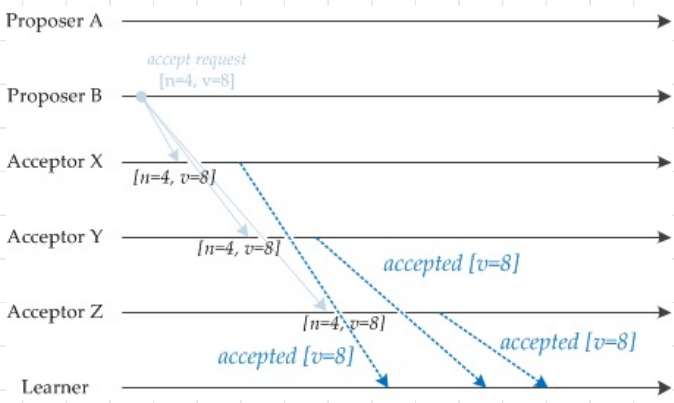
③ Acceptor 2 ignores proposal from A



④ Accept requests sent (accept from A is ignored)



⑤ Send accept to learners



LEADER ELECTION

- Upon leader failure, choose new leader from non-faulty processes
- Any process can call for an election (at most one election called per process at any given time)
- Result of election independent of who proposes it
- Liveness condition: every node eventually enters a state in {elected, not_elected}
- Safety condition: only one node can enter elected state and eventually becomes leader

Formal Election Problem

- One run of election algorithm must guarantee

  1. **Safety**: from all non-faulty processes p, one non-faulty process q with the best attribute value is elected as leader or election terminates with NULL

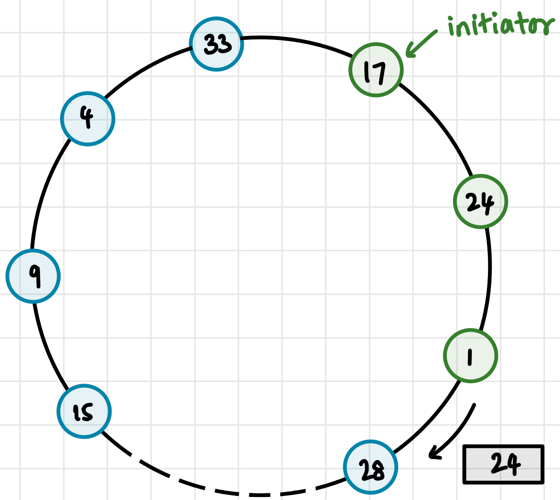
  2. **Liveness**: all non-faulty processes eventually enter a state in {elected, non-elected}

- Attribute values: fastest CPU, most disk space, priority, most no of files

Ring Election Algorithm

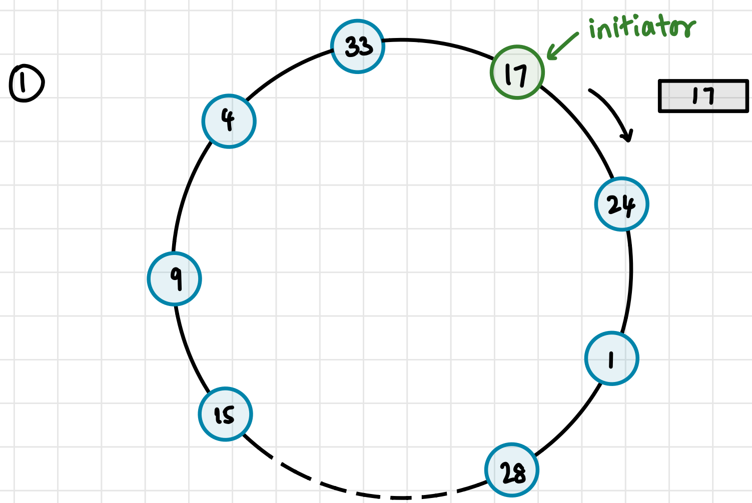
- N processes in a logical ring s.t. every node communicates only with its neighbours

- All messages sent clockwise

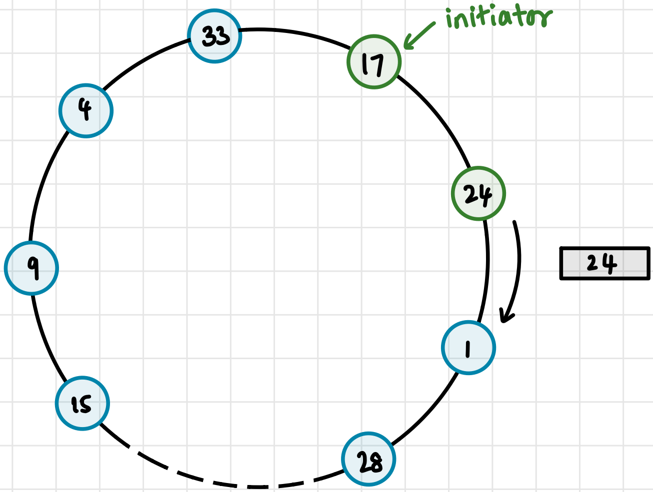


- If a process p_i discovers that coordinator has failed, initiates election message with p_i's ID — initator
- If a process p_j receives election message, compares p_i of message with its own ID p_j
  - If incoming $p_i > $ its own p_j, forwards message
  - If incoming $p_i < p_j$ and it hasn't yet forwarded an election message, overwrites incoming attribute with p_j and forwards
  - If $p_i == p_j$, p_j's attribute must be greatest (one complete round) and p_j is the new coordinator
- If elected, p_j sends an "elected" message to its cw neighbour, with its process ID p_j
- If a process p_k receives an "elected" message from p_j
  - sets its variable $elected_k$ as ID of p_j's message
  - forwards message (unless it is p_j itself, to prevent infinite messages)

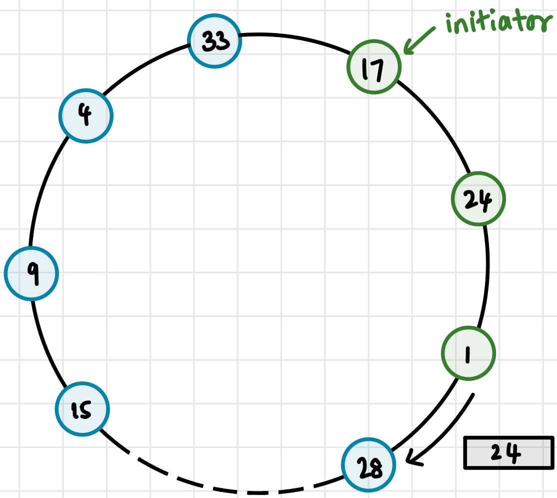
Q: Perform ring election if 17 is the initiator

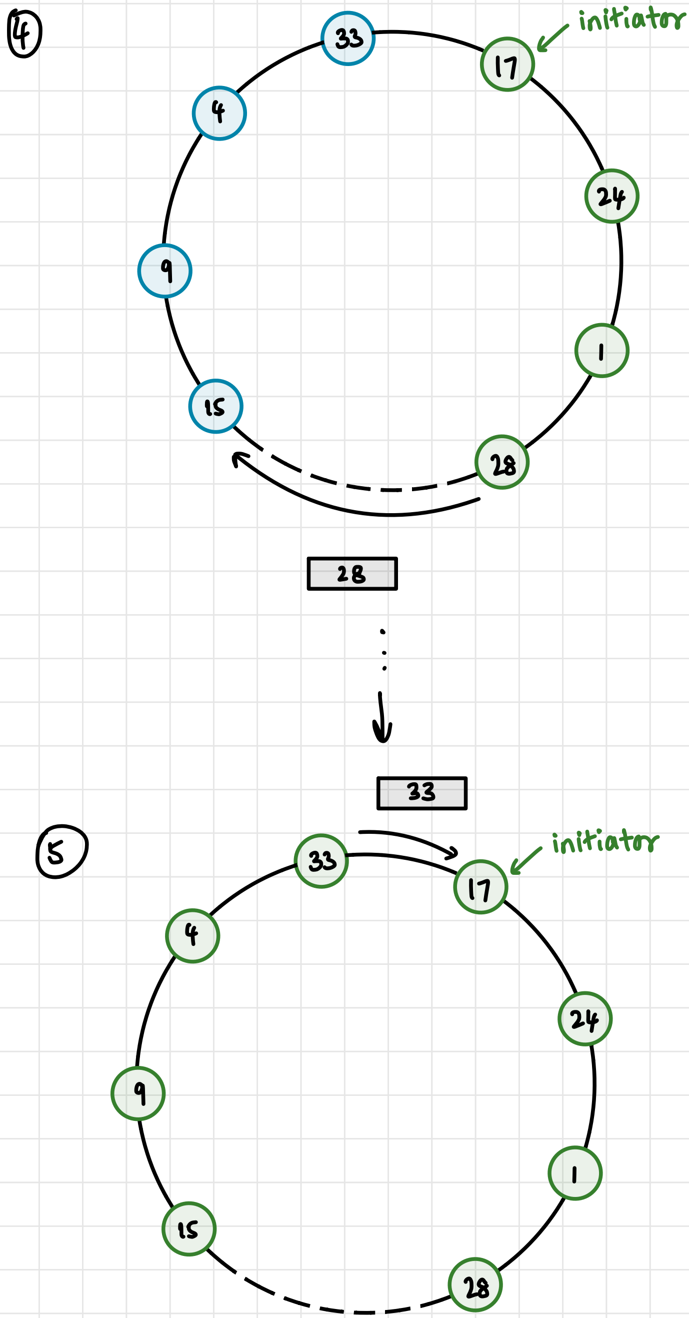


②



③





must do another round of forwarding until 33 receives a message with attr = 33

must then do another round of forwarding elected messages

Time Complexity

- Worst-case: initiator's ccw neighbour has highest attr
- N nodes in ring
- N-1 messages until highest node receives message
  N messages for highest node to receive its own message
  N messages to circulate newly elected message
- Total: 3N-1 = O(N)

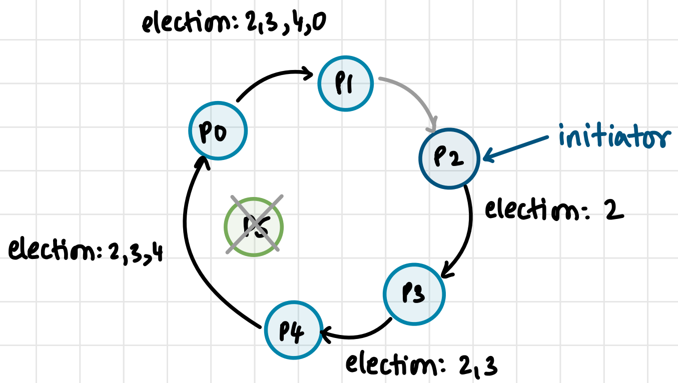
Ring Election with Failures

- If highest node fails, algorithm never terminates (no liveness)
- Modified ring election
  - Instead of p_j replacing p_i's attribute if $p_j > p_i$, p_j appends its attribute to the message (irrespective of whether $p_j > p_i$)
  - Bypass failed processes
  - Once reaches initiator, elects process with highest attr value
  - Sends "coordinator" message with ID of newly elected leader and every process appends its ID to end of message after locally storing newly elected leader
  - Once "coordinator" message received at initiator, election is terminated if elected ID is on ID list

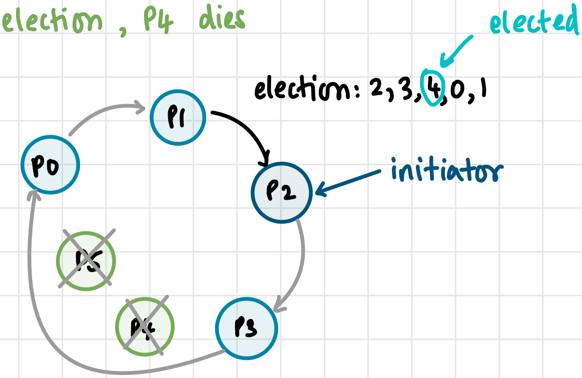
- If not, algorithm repeated

Example

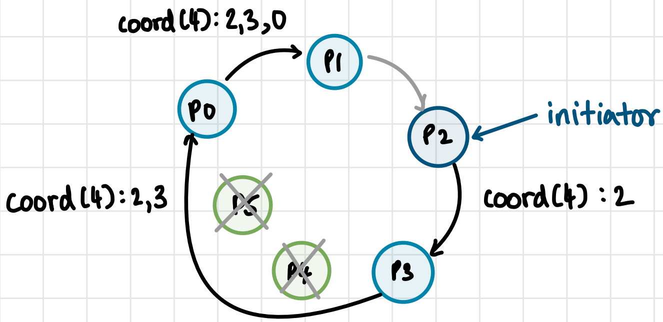
1. P2 initiates election



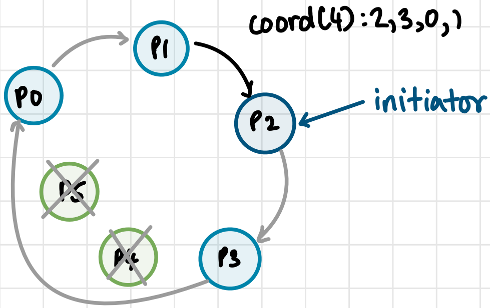
2. P2 receives election, P4 dies



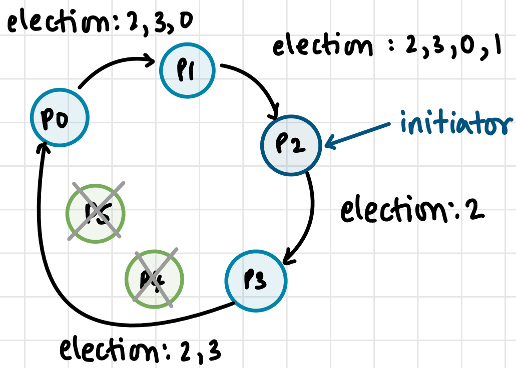
3. P2 selects 4 and announces results



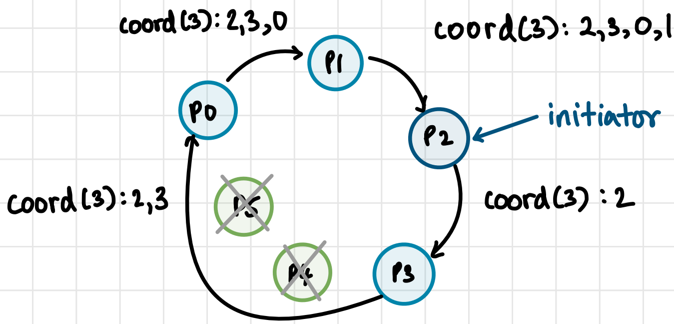
4. P2 receives coord(4) but 4 is not on the list



5. P2 re-initiates election



6. P2 finally elects P3

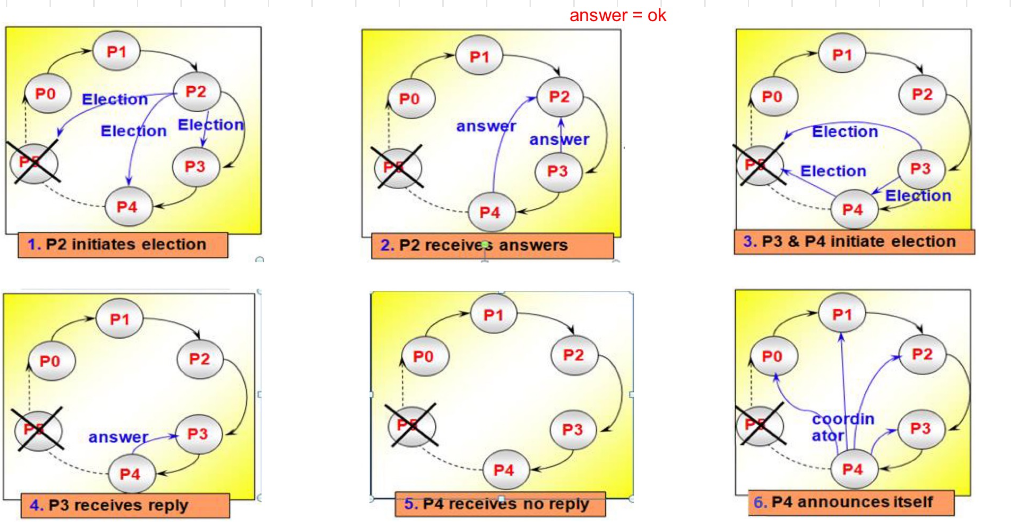


Bully Election Algorithm

- System where every process can send message to every other process
- Three types of messages
  1. election: sent to announce an election
  2. answer: sent in response to election message
  3. coordinator: announce identity of new leader
- When leader fails, if a process knows that it has the next-highest attribute, it elects itself as leader and sends a coordinator message to all other processes with lower attrs
- If process does not know, it initiates election with election message and sends to processes with higher attrs only
- Then it awaits for answer
  - If none received within timeout, elects itself as leader and sends coordinator message
  - Else, wait for coordinator message
  - If no coordinator message received within timeout, start a new election run
- If process receives election message, sends answer message and begins a new election run (unless it already has done before)
- If process p_i receives coordinator message, sets variable $elected_i$ to be ID of the coordinator

Timeout Values

- Assume one-way message transmission time is known (T)
- First timeout value (process that initiated election waits for response) ≈ 2T + (processing time) ≈ 2T
- Second timeout (process receives election and sends answer/disagree message and starts new election) – worst case turnaround time
- Synchronous assumptions
  - All messages sent in T_{trans} time (arrive)
  - Reply dispatched in $T_{process}$ time after receipt
  - No response in $2T_{trans} + T_{process}$ → process assumed to be faulty
- Other assumptions
  - All processes are aware of ID's of all other processes (their attributes)



1. We start with 5 processes, which are connected to each other. Process 5 is the leader, as it has the highest number.

2. Process 5 fails.

3. Process 2 notices that Process 5 does not respond. Then it starts an election, notifying those processes with ids greater than 2.

4. Then Process 3 and Process 4 respond, telling Process 2 that they'll take over from here.

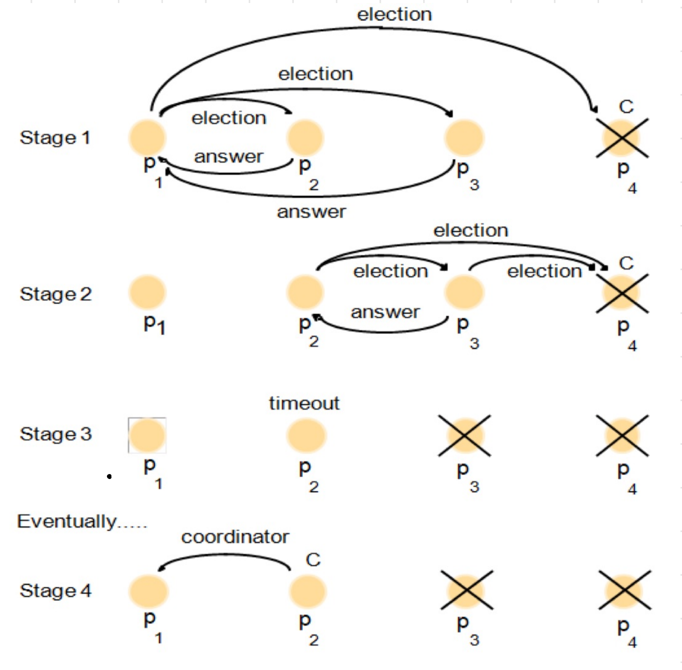
5. Process 3 sends election messages to the Process 4 and Process 5.

6. Only Process 4 answers to process 3 and takes over the election.

7. Process 4 sends out only one election message to Process 5.

8. When Process 5 does not respond Process 4, then it declares itself the winner.

Failures During Election Runs

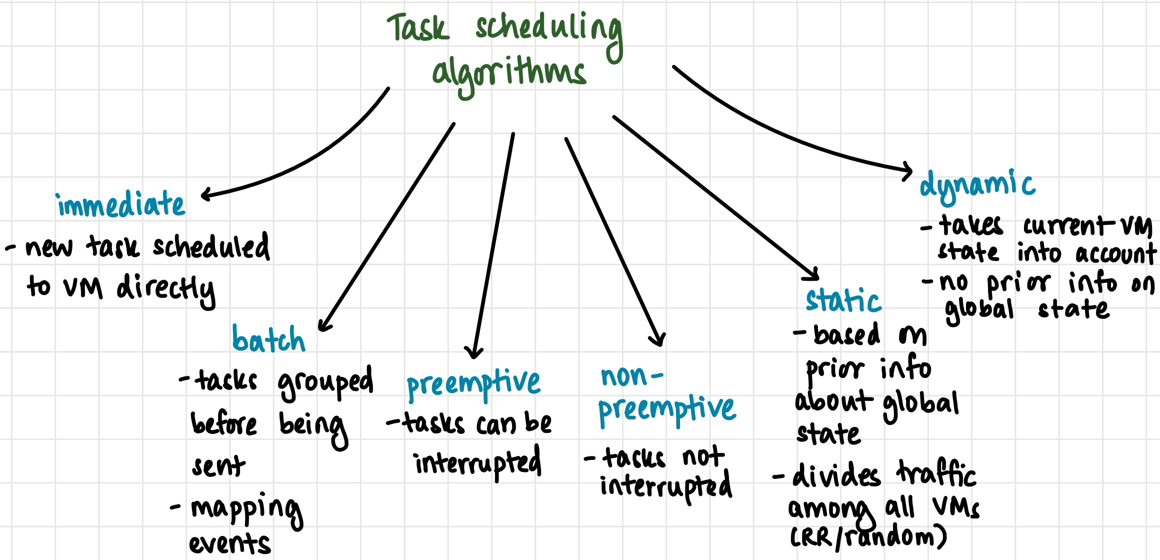


Time Complexity

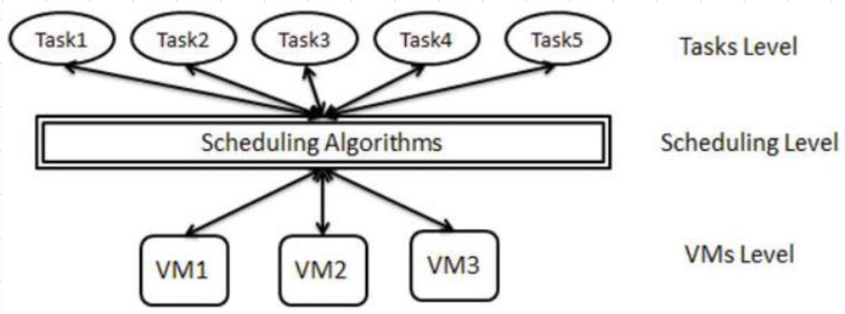
- Worst-case: when failure detected by lowest process
- Node sends election to N-1 nodes; N-1 responses (answer)
- Each of the N-1 processes p_i sends to N-1-i processes (P1 sends to N-2, P2 to N-3, ..., P_{N-2} to 1, P_{N-1} to 0)
  - Assuming N processes P_0 to P_N
- $O(N^2)$ complexity
- Turnaround time ≈ 5 message transmissions

https://www.cs.colostate.edu/~cs551/CourseNotes/Synchronization/BullyExample.html

TASK SCHEDULING



Levels of Task Scheduling



1. Tasks level
   - set of tasks/cloudlets sent by cloud users
   - Required for execution

2. Scheduling level
   - Mapping tasks to compute resources
   - Makespan: overall completion time for all tasks

3. VM level
   - Set of VMs

Static Task Scheduling Algorithms

1. FCFS
2. SJF
3. MAX-MIN

1. First Come, First Serve

- Order based on arrival time

Q: Assume 6 VMs with properties as shown (MIPS - million IPS) and tasks with following lengths. Apply FCFS.

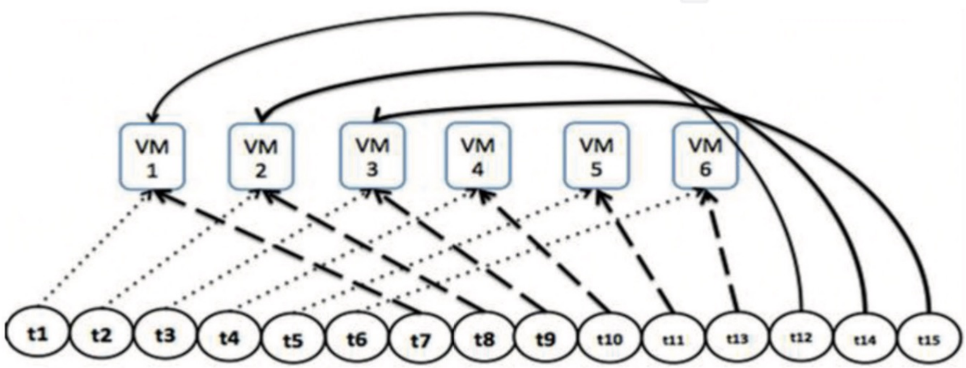
Task	Length
t1	100000
t2	70000
t3	5000
t4	1000
t5	3000
t6	10000
t7	90000
t8	100000
t9	15000
t10	1000
t11	2000
t12	4000
t13	20000
t14	25000
t15	80000

Assume we have six VMs with different properties based on tasks size:

VM list = {VM1, VM2, VM3, VM4, VM5, VM6}.

MIPS of VM list = {500, 500, 1500, 1500, 2500, 2500}.

- Note: tasks are assigned to VM and must wait for the prev task to execute
- Dotted - first, dashed - second, solid - third



Task	ET	Waiting time		
t1	200	VM1		
t2	140	VM2		
t3	3.33	VM3		
t4	0.66	VM4		
t5	1.2	VM5		
t6	4	VM6		
t7	180	Wait(200)	VM1	
t8	200	Wait(140)	VM2	
t9	10	Wait(3.33)	VM3	
t10	0.66	Wait(0.66)	VM4	
t11	0.8	Wait(1.2)	VM5	
t12	1.6	Wait(4)	VM6	
t13	40	Wait(380)		VM1
t14	50	Wait(340)		VM2
t15	53.33	Wait(13.33)		VM3

2. Shortest Job First

- Sort based on length
- Assume 6 VMs (like before)
- Can lead to starvation

Tasks	t4	t10	t11	t5	t12	t3	t6	t9	t13	t14	t2	t15	t7	t1	t8
lengths	1000	1000	2000	3000	4000	5000	10000	15000	20000	25000	70000	80000	90000	100000	100000

Task	ET	Waiting time		
t4	2	VM1		
t10	2	VM2		
t11	1.33	VM3		
t5	2	VM4		
t12	1.6	VM5		
t3	2	VM6		
t6	20	Wait(2)	VM1	
t9	30	Wait(2)	VM2	
t13	13.33	Wait(1.33)	VM3	
t14	16.66	Wait(2)	VM4	
t2	28	Wait(1.6)	VM5	
t15	32	Wait(2)	VM6	
t7	180	Wait(22)		VM1
t1	200	Wait(32)		VM2
t8	66.66	Wait(14.66)		VM3

3. Max-Min

- Tasks sorted based on completion time
- Long tasks - high priority - VMs with shortest execution time

Tasks	t1	t8	t7	t15	t2	t14	t13	t9	t6	t3	t12	t5	t11	t10	t4
lengths	100000	100000	90000	80000	70000	25000	20000	15000	10000	5000	4000	3000	2000	1000	1000

Task	ET	Waiting time		
t1	40	VM6		
t8	40	VM5		
t7	60	VM4		
t15	53.33	VM3		
t2	140	VM2		
t14	50	VM1		
t13	8	Wait(40)	VM6	
t9	6	Wait(40)	VM5	
t6	6.66	Wait(60)	VM4	
t3	3.33	Wait(53.33)	VM3	
t12	8	Wait(140)	VM2	
t5	6	Wait(50)	VM1	
t11	0.8	Wait(48)		VM6
t4	0.4	Wait(46)		VM5
t10	0.67	Wait(66.67)		VM4

DISTRIBUTED LOCKING

- Quorum: min no. of votes for acceptance

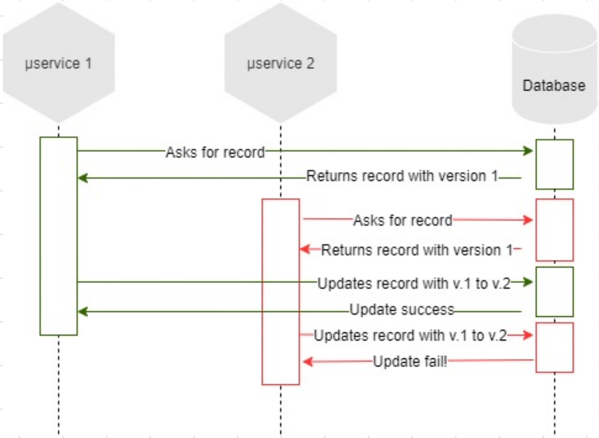
- Reasons to lock
  1. Efficiency
  2. Correctness

- Features of distributed locks
  1. Mutual exclusion
  2. Deadlock-free
  3. Consistency

Types of Distributed Locks

1. Optimistic

- Do not block potentially dangerous events
- Hope for the best

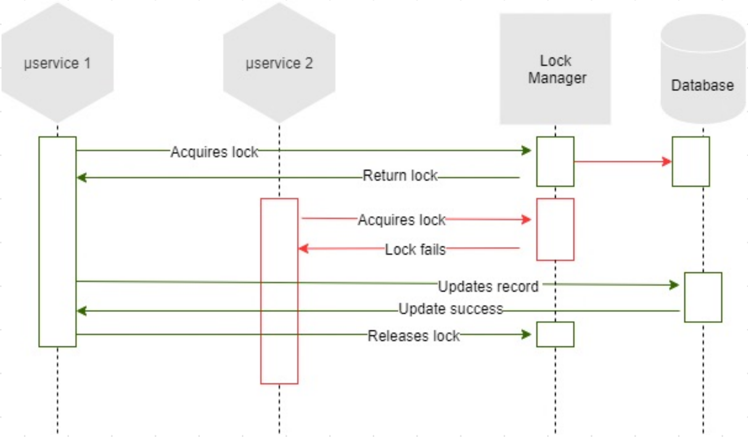


uses version numbers

if version mismatch, fail

2. Pessimistic

- Block access to resource before operating
- Release when done



Implementing Distributed Locking

For example, say you have an application in which a client needs to update a file in shared storage (e.g. HDFS or S3). A client first acquires the lock, then reads the file, makes some changes, writes the modified file back, and finally releases the lock. The lock prevents two clients from performing this read-modify-write cycle concurrently, which would result in lost updates. The code might look something like this:

```
// THIS CODE IS BROKEN
function writeData(filename, data) {
    var lock = lockService.acquireLock(filename);
    if (!lock) {
        throw 'Failed to acquire lock';
    }

    try {
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.release();
    }
}
```

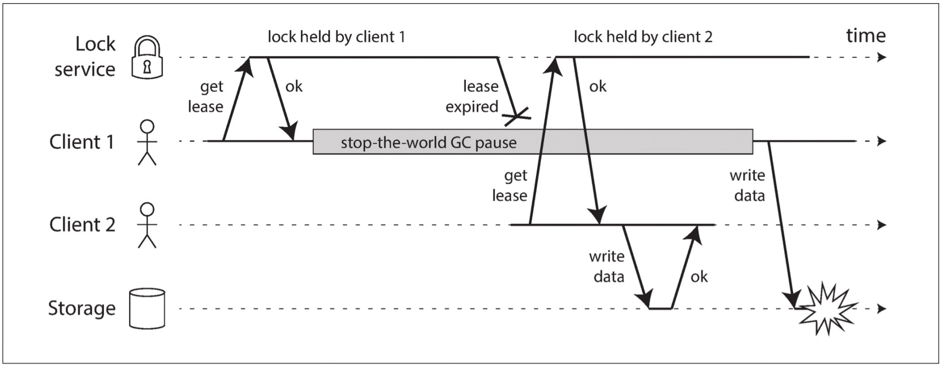


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

- HBase used to have this problem (due to GC pauses)

Distributed Locking with Fencing

- Use fencing tokens with every write request to the storage service
- Fencing token: no. that increases every time a client acquires a lock

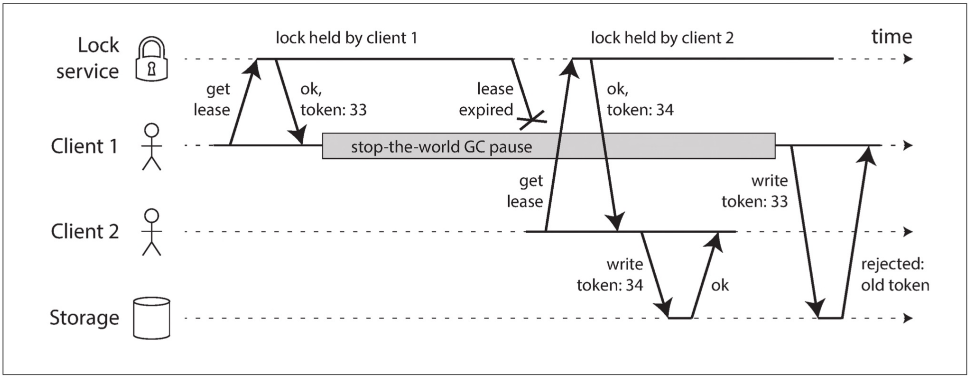
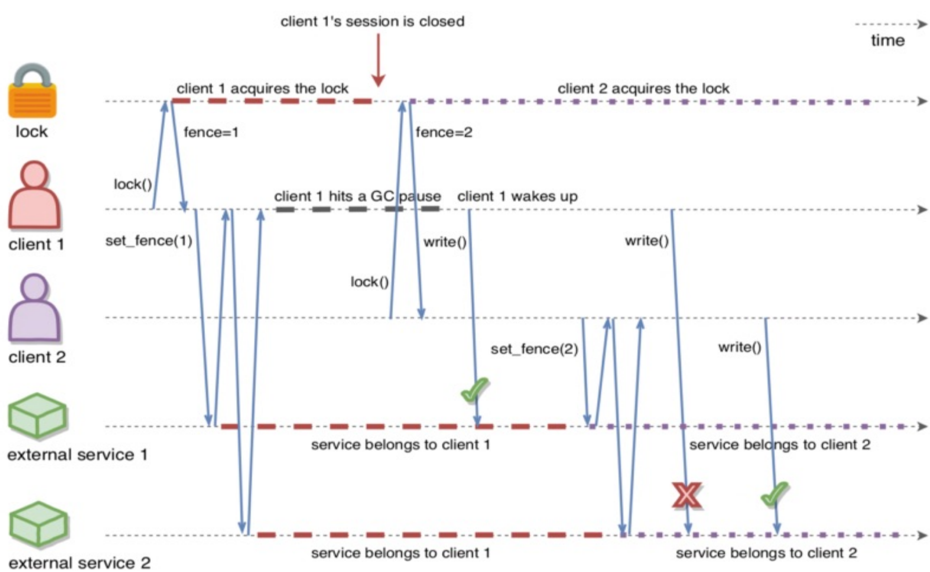


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.



Distributed Lock Manager

- Google Chubby
- ZK
- Redis

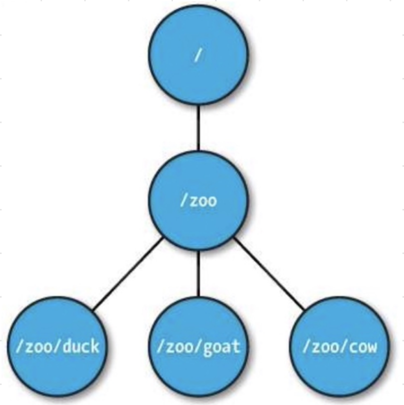
Zookeeper

- Distributed coordination service

- Features
  1. Update node status
  2. Managing cluster
  3. Naming service
  4. Automatic failure recovery

Data Model

- Hierarchical namespace
- znodes : data & children
- Tree kept in memory
- Like file system
- Small amounts of data – coordination data, status info etc



Types of Znodes

1. Persistent

- Need to be deleted explicitly by client
- Permanent (even after session terminated)

2. Ephemeral

- Automatically deleted when session that created it ends
- Used to detect termination of client
- Can set up watches
- Not allowed to have children

3. Sequence

- Append monotonically increasing counter to end of path
- Both persistent & ephemeral

Watches

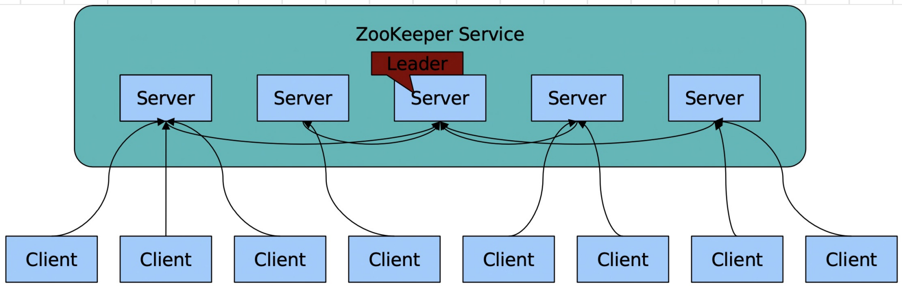
- Clients get notified when znode modified
- Too many watches - herd effect

Data Access

- Access Control List for each node

Zookeeper Servers

- Leader elected at startup
- Only followers service clients



- All servers: one copy of the data tree (memory)
- Transaction logs: persistent store
- Changes to znodes → added to transaction logs
- One server per client until connection breaks/ends
- Zookeeper Atomic Broadcast (ZAB) protocol

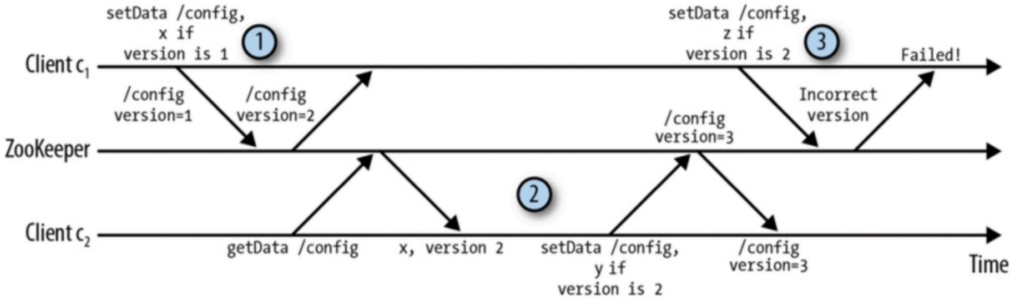
Reads
- Processed locally at server

Writes
- Req forwarded to leader
- Leader gets majority consensus
- Response generated

Watches
- B/w client and single server
- On a znode

ZK Operations

Operation	Type
create	Write
delete	Write
exists	Read
getChildren	Read
getData	Read
setData	Write
getACL	Read
setACL	Write
sync	Read



1 Client c_1 writes the first version of /config.
2 Client c_2 reads /config and writes the second version.
3 Client c_1 tries to write a change to /config, but the request fails because the version does not match.

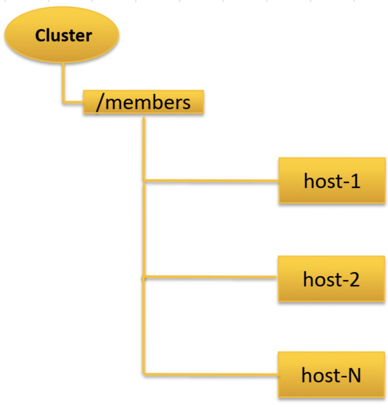
Hadoop on Demand

- Slides

Usage of ZK

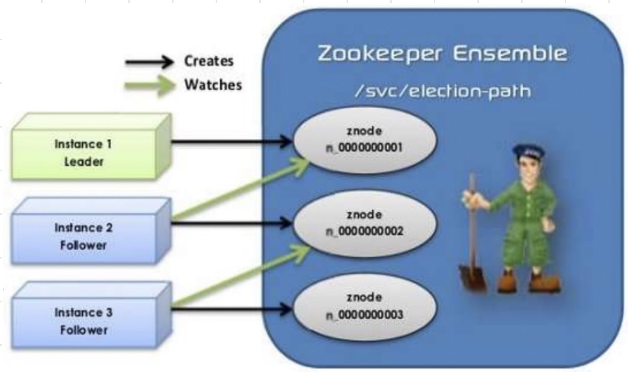
1. Configuration Management

- Keep track of nodes in cluster (clients)
- create /members/host-${i} as ephemeral nodes
- Watch on /members



2. Leader election

- All participants of election process create ephemeral-sequential node on election path
- /svc/election-path
- Leader: smallest seq. no
- Followers: listen to node with next lowest seq. no



3. Distributed Exclusive Locks

- Queue of clients waiting for a lock as ephemeral nodes under /cluster/_locknode_
- Watch on prev host
- Client with least ID holds lock
- Herd effect

```
ZK
|---Cluster
   +---config
   +---memberships
   +---_locknode_
      +---host1-3278451
      +---host2-3278452
      +---host3-3278453
      +--- ...
      \---hostN-3278XXX
```