Blockchain Unit 2

B.tech (Dr. A.P.J. Abdul Kalam Technical University)

UNIT 2

Consensus:

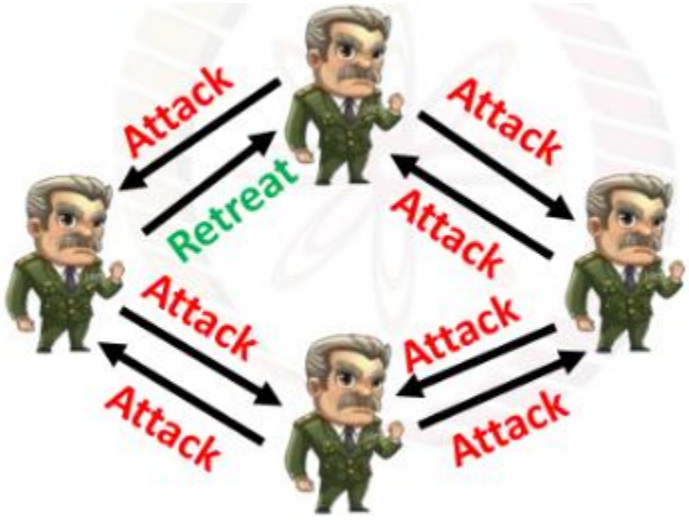
- A procedure to reach in a common agreement in a distributed or decentralized multi-agent platform
- Important for a message passing system

Why Need Consensus?

Reliability and fault tolerance in a distributed system
– Ensure correct operations in the presence of faulty individuals
• **Example:**
– Commit a transaction in a database
– State machine replication
– Clock synchronization

Why Consensus Can be Difficult in Certain Scenarios?

In message passing system, node behave maliciously



Distributed Consensus

- If there is no failure, it is easy and trivial to reach in a consensus
– Broadcast the personal choice to all
– Apply a choice function, say the maximum of all the values

Different types of distributed consensus fault:

There can be various types of faults in a distributed system.
• Crash Fault: A node suddenly crashes or becomes unavailable in the middle of a communication
• Network or Partitioned Faults: A network fault occurs (say the link failure) and the network gets partitioned
• Byzantine Faults: A node starts behaving maliciously

Properties of Distributed Consensus

- **Termination:** Every correct individual decides some value at the end of the consensus protocol

• **Validity:** If all the individuals proposes the same value, then all correct individuals decide on that value

• **Integrity:** Every correct individual decides at most one value, and the decided value must be proposed by some individuals

• **Agreement:** Every correct individual must agree on the same value

Synchronous vs Asynchronous Systems

Synchronous Message Passing System: The message must be received within a predefined time interval

– Strong guarantee on message transmission delay

• **Asynchronous Message Passing System:** There is no upper bound on the message transmission delay or the message reception time

– No timing constraint, message can be delayed for arbitrary period of times

Correctness of a Distributed Consensus Protocol

- **Safety:** Correct individuals must not agree on an incorrect value
  - Nothing bad happend

• Liveliness (or Liveness): Every correct value must be accepted eventually

- Something good eventually happens

Why Do We Require Consensus in Bitcoin Network?

Bitcoin is a peer-to-peer network

• Alice broadcast a transaction in this peer-to-peer network

• All the nodes in this network need to agree on the correctness of this transaction

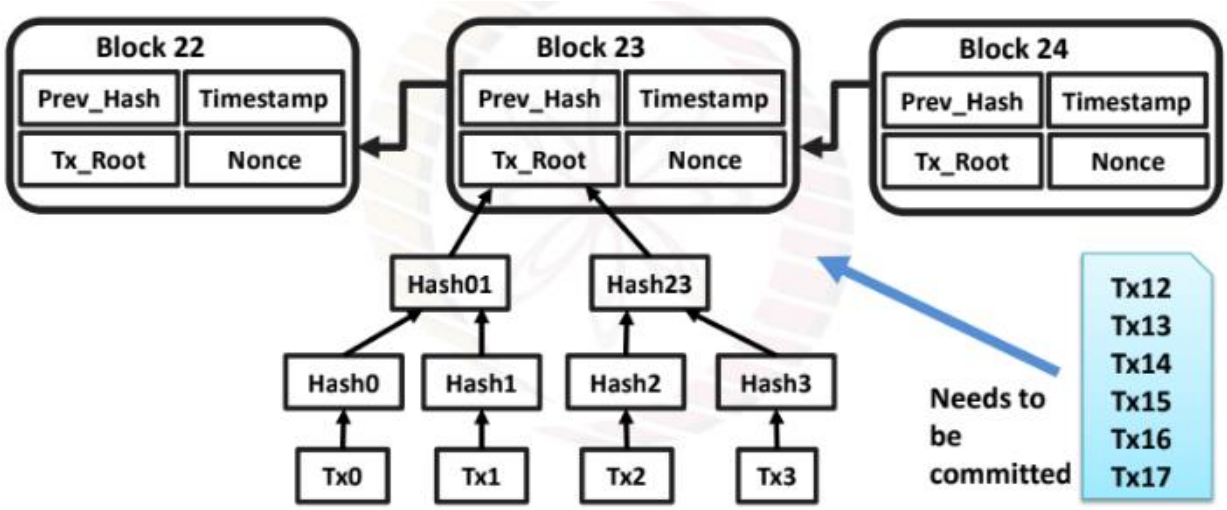
- A node does not know all the peers in the network – this is an open network

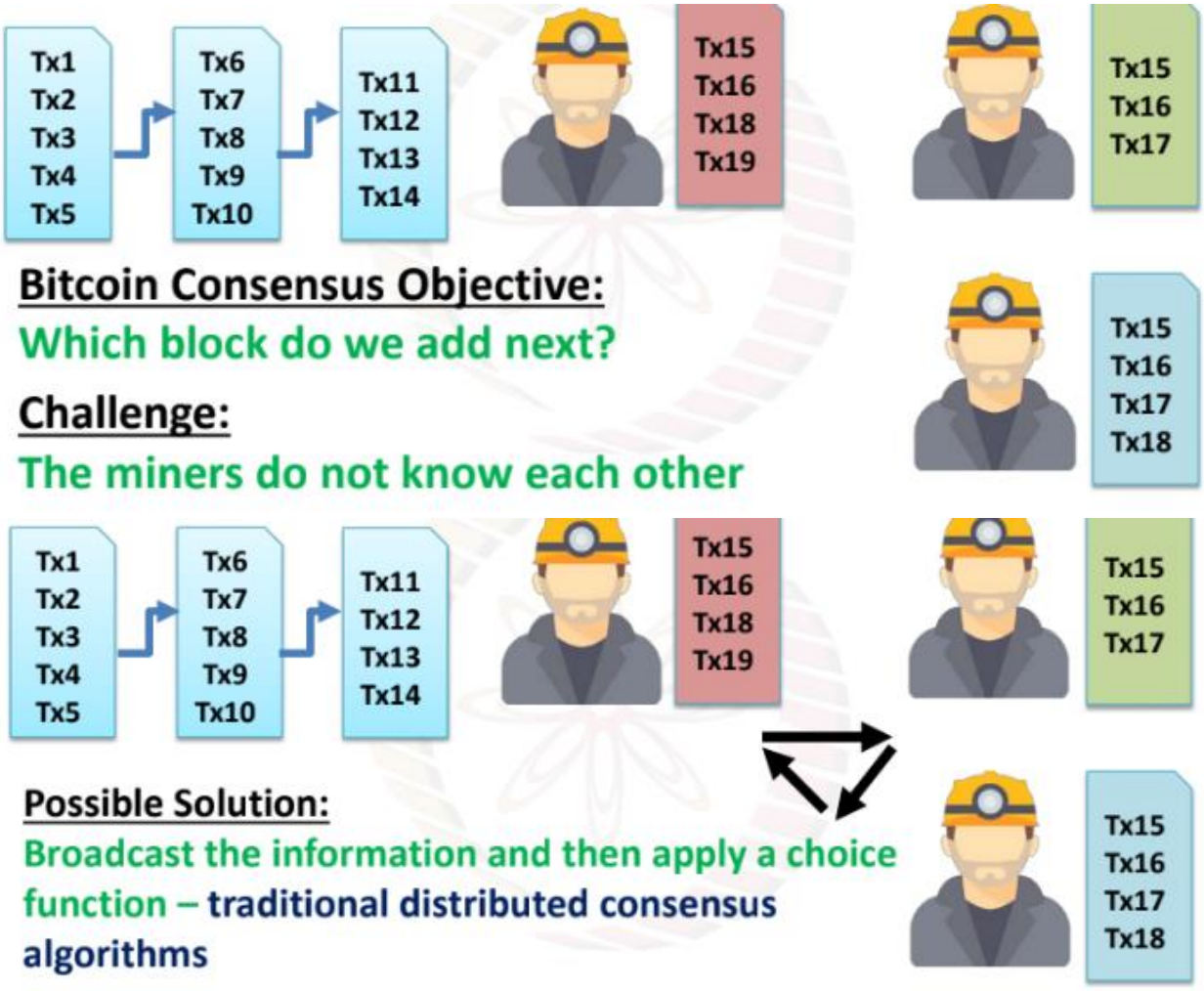
• Some nodes can also initiate malicious transactions

Consensus in a Bitcoin

- Every node has block of transactions that has already reached into the consensus (block of committed transactions)

• The nodes also has a list of outstanding transactions that need to be validated against the block of committed transactions





May not be Feasible:

- You do not have a global clock! How much time will you wait to hear the transactions
- Remember the impossibility result

Observation - 1:

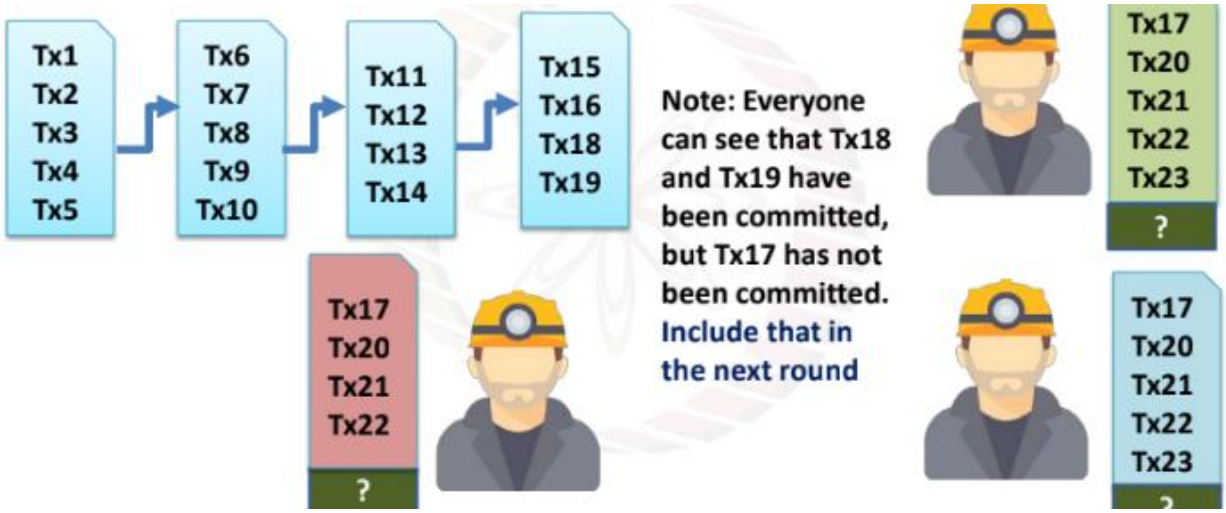
• Any valid block (a block with all valid transactions) can be accepted, even if it is proposed by only one miner

Observation - 2:

• The protocol can work in rounds
• Broadcast the accepted block to the peers
• Collect the next set of transactions

Solution:

• Every miner independently tries to solve a challenge
• The block is accepted for the miner who can prove first that the challenge has been solved



Permissionless Blockchain :

The permissionless or open model of blockchain – any user can join the network and participate in transactions

– Bitcoin is developed on this principle

• The blockchain provides the backbone of the permissionless digital currency

– Provides a decentralized architecture

– Tamper-proof through hash-chain

• Miners ensures the consensus in the system

Different permisionless blockchain consensus algorithm are as follows,

1. Proof of Work (Pow)
2. Proof of Stake(PoS)
3. Proof of Burn(PoB)
4. Proof of Elapsed Time(PoET)

1. Proof of Work (PoW)

PoW works as an Asymmetry,

– The work must be moderately hard, but feasible for the service requester

– The work must be easy check for the service provider

• Service requesters will get discouraged to forge the work, but service providers can easily check the validity of the work

Cryptographic Hash as the PoW

Use the puzzle friendliness property of cryptographic hash function as the work

– Given XX and YY, find out k, such that Y = H(X||k)

– It is difficult (but not infeasible) to find such k

– However, once you have a k, you can easily verify the challenge

• Used in Hashcash, a proof of work that can be added with an email as a “good-will” token

Bitcoin Proof of Work System

- Most implementations of Bitcoin PoW use double SHA256 hash function

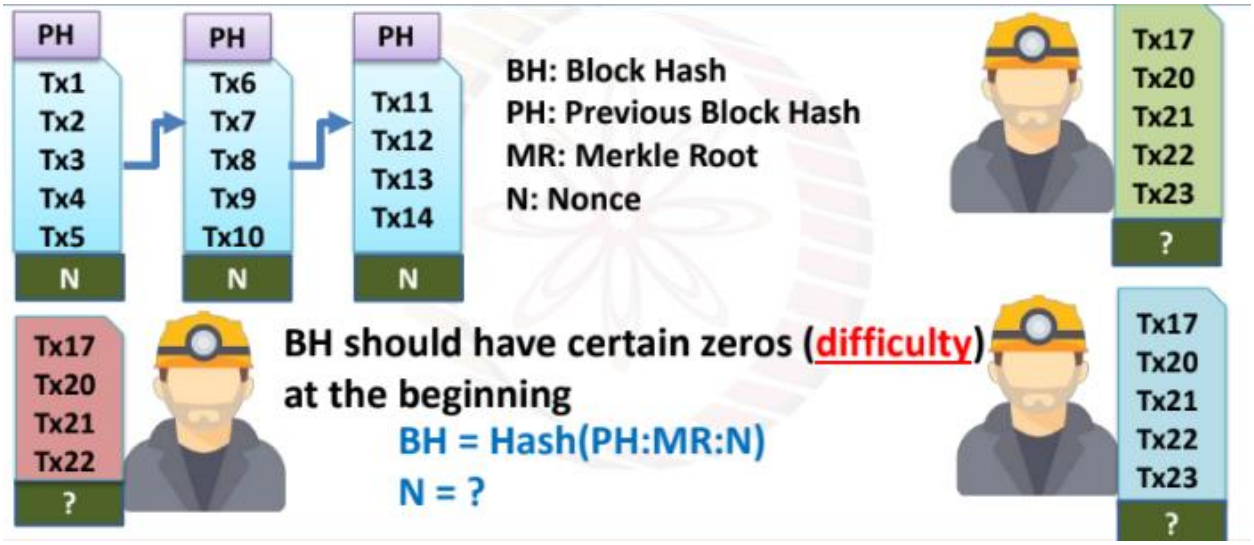
• The miners collect the transactions for 10 minutes (default setup) and starts mining the PoW

• The probability of getting a PoW is low – it is difficult to say which miner will be able to generate the block

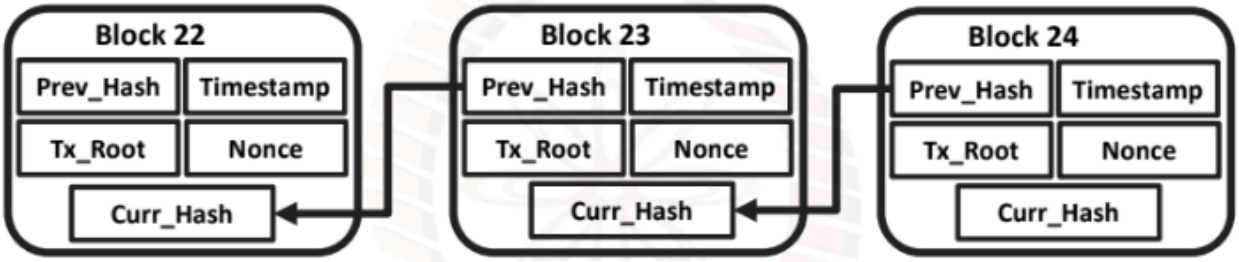
– No miner will be able to control the bitcoin network single handedly
Based on Hashcash PoW system

– The miners need to give a proof that they have done some work, before proposing a new block

– The attackers will be discouraged to propose a new block, or make a change in the existing blocks



Tampering PoW Blockchain



The blockchain together contain a large amount of work

– The attacker needs to perform more work to tamper the blockchain

– This is difficult with the current hardware

Solving the Double Spending Problem

The attack: Successful use of the same fund twice

– A transaction is generated with BTC10 to both Bob and C at the same time

• The solution:

– The transactions are irreversible (computationally impractical to modify)

– Every transaction can be validated against the existing blockchain

Monopoly Problem (Limitation of PoW)

- PoW depends on the computing resources available to a miner
  – Miners having more resources have more probability to complete the work

• Monopoly can increase over time (Tragedy of the Commons)
  – Miners will get less reward over time
  – Users will get discouraged to join as the miner
  – Few miners with large computing resources may get control over the network

(2) Proof of Stake (PoS)

- Handling Monopoly and Power Consumption
- **PoW vs PoS**
  – PoW: Probability of mining a block depends on the work done by the miner
  – PoS: Amount of bitcoin that the miner holds – Miner holding 1% of the Bitcoin can mine 1% of the PoS blocks.
- **Provides increased protection**
  – Executing an attack is expensive, you need more Bitcoins
  – Reduced incentive for attack – the attacker needs to own a majority of
- Bitcoins – an attack will have more affect on the attacker

(3) Proof of Burn (PoB)

• Miners should show proof that they have burned some coins
  – Sent them to a verifiably un-spendable address
  – Expensive just like PoW, but no external resources are used other than the burned coins

- **PoW vs PoB** – Real resource vs virtual/digital resource
- PoB works by burning PoW mined cryptocurrencies

Difference between PoW, PoS, PoB

PoW	PoS	PoB
• Do some work to mine a new block	• Acquire sufficient stake to mine a new block	• Burn some wealth to mine a new block
• Consumes physical resources, like CPU power and time	• Consumes no external resource, but participate in transactions	• Consumes virtual or digital resources, like the coins
• Power hungry	• Power efficient	• Power efficient

(4) Proof of Elapsed Time (PoET)

- Proposed by Intel, as a part of Hyperledger Sawtooth – a blockchain platform for building distributed ledger applications

• Basic idea:
  – Each participant in the blockchain network waits a random amount of time
  – The first participant to finish becomes the leader for the new block

- **How will one verify that the proposer has really waited for a random amount of time?**
  - Utilize special CPU instruction set – Intel Software Guard Extension (SGX)
    – a trusted execution platform
    – The trusted code is private to the rest of the application
    – The specialized hardware provides an attestation that the trusted code has been set up correctly

Mining Bitcoins

- Find a nonce to make the new block valid
- Broadcast the new block – everybody accepts it if it is a part of the main chain

• Earn the reward for participating in the mining procedure

Mining Difficulty:

• A measure of how difficult it is to fond a hash below a given target
– Bitcoin network has a global block difficulty
– Mining pools also have a pool-specific share difficulty
• The difficulty changes for every 2016 blocks
– Desired rate – one block each 10 minutes
– Two weeks to generate 2016 blocks
– The change in difficulty is in proportion to the amount of time over or under two weeks the previous 2016 blocks took to find (en.bitcoin.it)
• Compute the following for every two weeks
current_difficulty = previous_difficulty * (2 weeks in milliseconds)/(milliseconds to mine last 2016 blocks)

Setting the Difficulty of Mining

• The hash is a random number between 0 and 2256-1
– To find a block, the hash must be less than a given target
• The offset for difficulty 1 is 0xffff * 2208
• The offset for difficulty D is 0xffff * 2208/D
• The expected number of hashes we need to calculate to find a block with difficulty D is (D * 2256) / (0xffff * 2208)

Permissioned Blockchain

A blockchain architecture where users are authenticated apriory
• Users know each other
• However, users may not trust each other – Security and consensus are still required.
• Run blockchain among known and identified participants

Design Limitations of Permissioned Blockchain

1) Sequential Execution

– Execute transactions sequentially based on consensus
– Requests to the application (smart contract) are ordered by the consensus, and executed in the same order
– This give a bound on the effective throughput – throughput is inversely proportional

– Can be a possible attack on the smart contract platform – introduce contract which will take long time to execute

2) Non-deterministic Execution

– Consider golang – iteration over a map may produce a different order in two executions

```
m := map[string]string{ "key1":"val1", "key2":"val2" };
for k, v := range m {
fmt.Printf("key[%s] value[%s]\n", k, v)
}
```

– Smart-contract execution should always needs to be deterministic; otherwise the system may lead to inconsistent states (many fork in the blockchain)

– Solution: Domain specific language (DSL) for smart contract

3) Execution on all nodes

– Generally, execute smart contracts at all nodes, and propagate the state to others – try to reach consensus

– Consensus: Propagate same state to all nodes, verify that the states match

– Do you have sufficient numbers of trusted nodes to validate the execution of smart contracts?

Use state machine replication – execute contract at a subset of nodes, and ensure that the same state is propagated to all the nodes

Why Distributed consensus in Permissioned algorithm?

- Reaching agreement in distributed computing
- Replication of common state so that all processes have same view
- Applications:

– Flight control system: E.g. Boeing 777 and 787

– Fund transferring system: Bitcoin and cryptocurrencies

– Leader election/Mutual Exclusion

So, no need of consensus in a single node process.

- **What about when there are two nodes?**
  - Network or partitioned fault, consensus cannot be reached

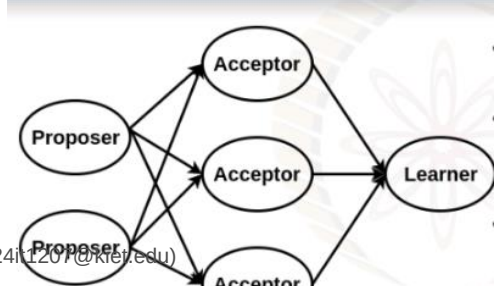
Different types of Consensus algorithm in Permissioned Blockchain

1) **Crash or Network Faults:**
   - PAXOS
   - RAFT

(2) **Byzantine Faults (including Crash or Network Failures):**
   - Byzantine fault tolerance (BFT)
   - Practical Byzantine Fault Tolerance (PBFT)

PaXoS

- First Consensus Algorithm proposed by L.
- Lamport in 1989



• Objective: choosing a single value under crash or network fault

• **System process**

– Making a proposal

– Accepting a value

– Handling Failures

Types of nodes:

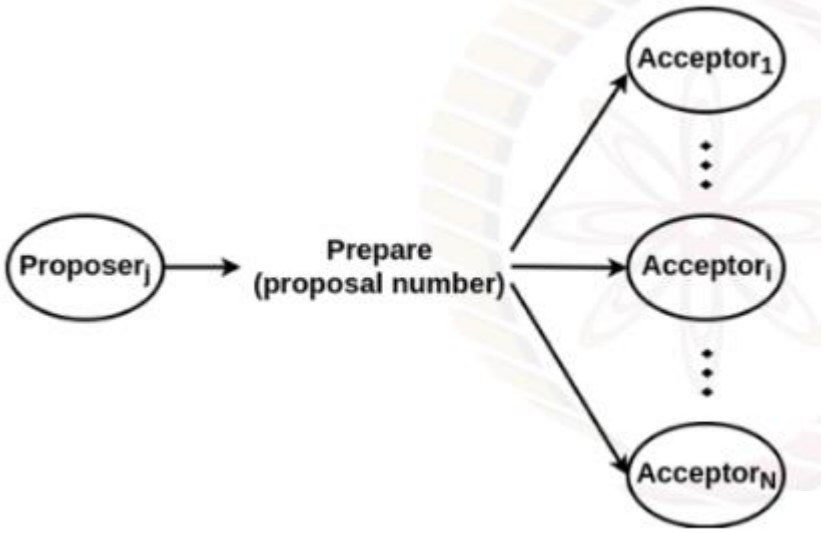
- **Proposer:** propose values that should be chosen by the consensus

• **Acceptor:** form the consensus and accept values

• **Learner:** learn which value was chosen by each acceptor

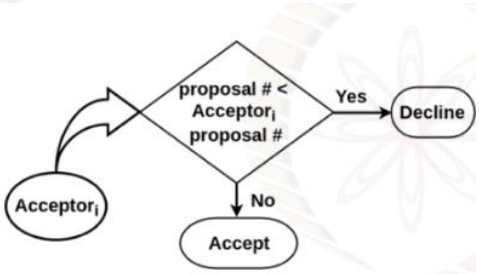
Making a Proposal: Proposer Process

proposal number: form a timeline, biggest number considered up-to-date

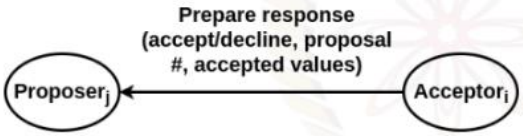


Acceptor Decision:

Each acceptor compares received proposal number with the current known values for all proposer's prepare message



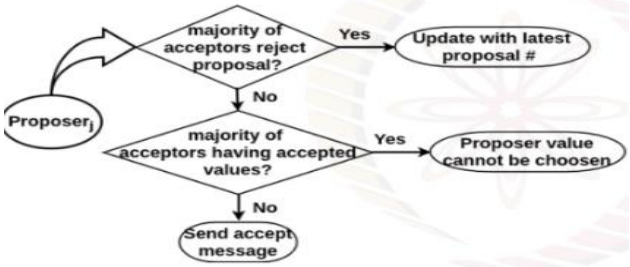
Acceptor's Message:



accept/decline: whether prepare accepted or not

• **proposal number:** biggest number the acceptor has seen

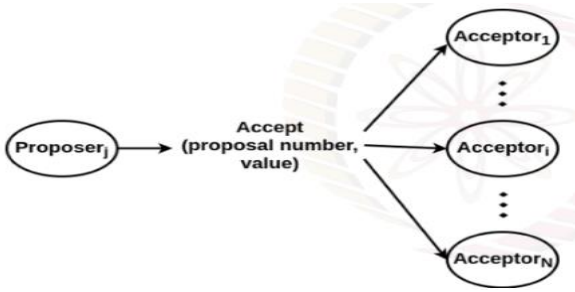
• **accepted values:** already accepted values from other proposer



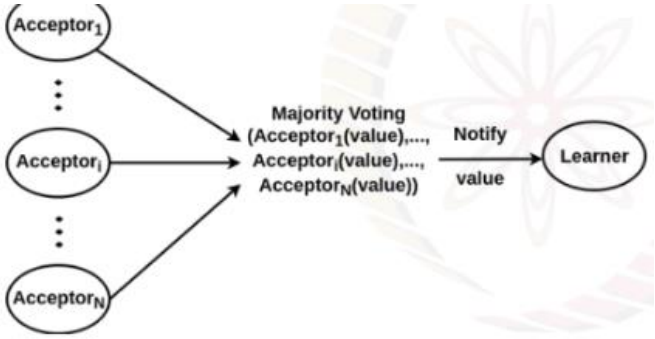
Accepting a Value: Proposer's Decision Making

Proposer receive a response from majority of acceptors before

Accepting a Value: Accept Message
proposal number:
same as prepare phase
value
• value: single value
proposed by proposer



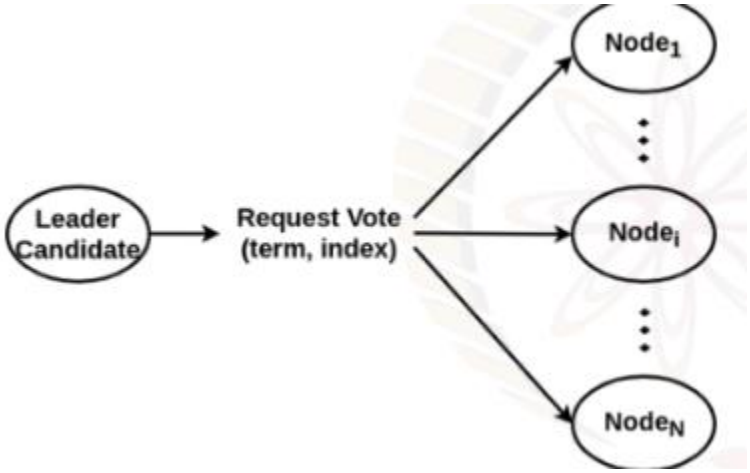
Accepting a Value: Notifying Learner
Each acceptor accept value from any of the proposer
• Notify learner the majority voted value



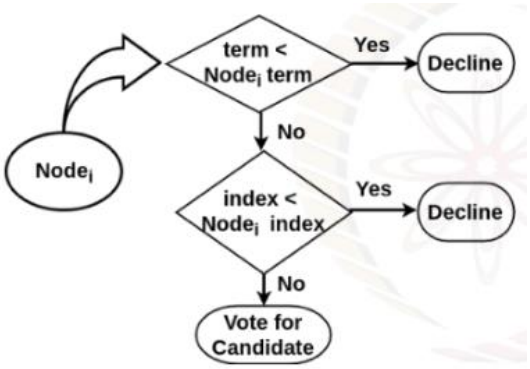
RAFT Consensus

Designed as an alternative to Paxos
• A generic way to distribute a state machine among a set of servers
– Ensures that every server agrees upon same series of state transitions
• Basic idea -
– The nodes collectively selects a leader; others become followers
– The leader is responsible for state transition log replication across the followers

(1) Electing the Leader: Voting Request
term: last calculated # known to candidate + 1
index: committed transaction available to the candidate



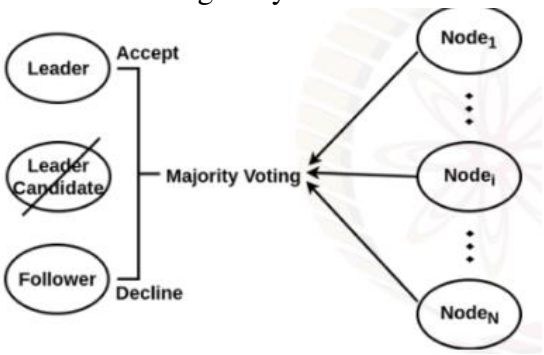
(2) Electing the leader: Follower Node's Decision Making



- Each node compares received term and index With corresponding current known values

(3) Electing the leader: Majority Voting

Use of Majority voting
– leader selection
– commit the log entry



Handling Failure in RAFT

- Failure of up to N/2 – 1 nodes does not affect the system due to majority voting

Byzantine Generals Problem

- Paxos and Raft can tolerate up to (N/2) − 1 number of crash faults
  • What if the nodes behave maliciously?

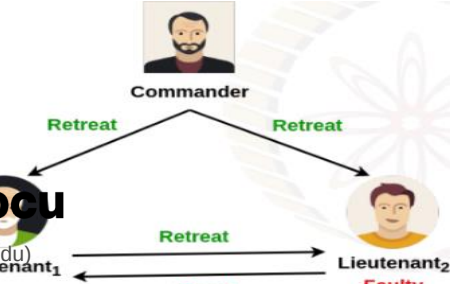
- **Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others.**

  The problem : To find an algorithm to ensure that loyal generals will reach a the desired agreement.

Three Byzantine Generals Problem

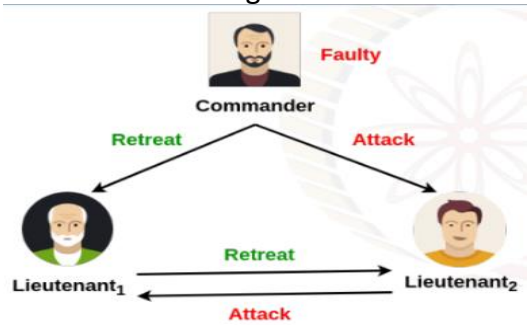
Case: 1 Lieutenant Faulty

Round1:



– Commander correctly sends same message to
Lieutenants

• **Round 2:**
– Lieutenant1 correctly echoes to Lieutenant2
– Lieutenant2 incorrectly echoes to Lieutenant1
Lieutenant1 received differing message
• By integrity condition, Lieutenant1 bound to decide on Commander message
• **What if Commander is faculty?**



Case: 2 Commander Faulty
Round 1:
– Commander sends differing message to Lieutenants
• **Round 2:**
– Lieutenant1 correctly echoes to Lieutenant2
– Lieutenant2 correctly echoes to Lieutenant1
• Lieutenant1 received differing message
• By integrity condition, both Lieutenants conclude with Commander's message
• This contradicts the agreement condition
• **No solution possible** for three generals including one faulty

Four Byzantine Generals Problem

Case 1: Lieutenant Faulty

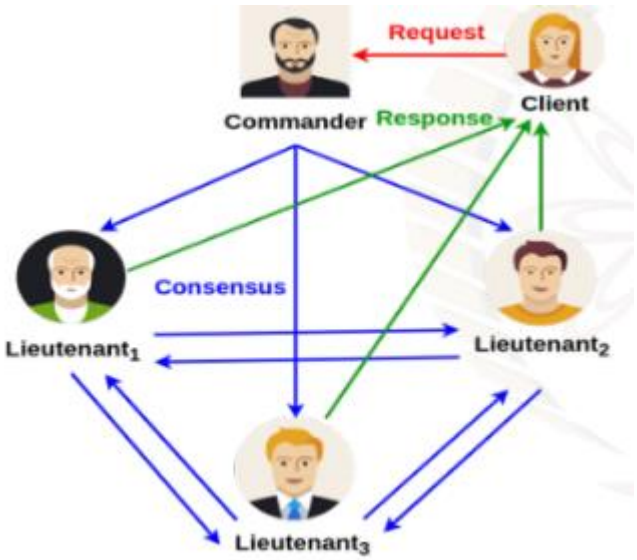
• **Round 1:**
– Commander sends a message to each of the Lieutenants
• Round 2:
– Lieutenant1 and Lieutenant3 correctly echo the message to others
– Lieutenant2 incorrectly echoes to others
• Lieutenant1 decides on
majority(Retreat,Attack,Retreat)= Retreat
• Lieutenant3 decides on
majority(Retreat,Retreat,Attack)= Retreat

Case 2: Commander Faulty
• **Round 1:**
– Commander sends differing message to Lieutenants
• **Round 2:**
– Lieutenant1, Lieutenant2 and Lieutenant3 correctly echo the message to others
• Lieutenant1 decides on
majority(Retreat,Attack,Retreat)= Retreat
• Lieutenant2 decides on
majority(Attack,Retreat,Retreat)= Retreat
• Lieutenant3 decides on
majority(Retreat,Retreat,Attack)= Retreat
In byzantine general problem,

N number of process with at most f faulty
• Receiver always knows the identity of the sender
• Fully connected
• Reliable communication medium
• Synchronous system

Practical Byzantine Fault Tolerant

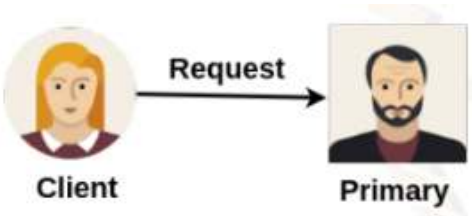
Why Practical?
– Ensures safety over an asynchronous network (not liveness!)
– Byzantine Failure
– Low overhead
• **Real Applications**
– Tendermint
– IBM's Openchain
– ErisDB
– Hyperledger



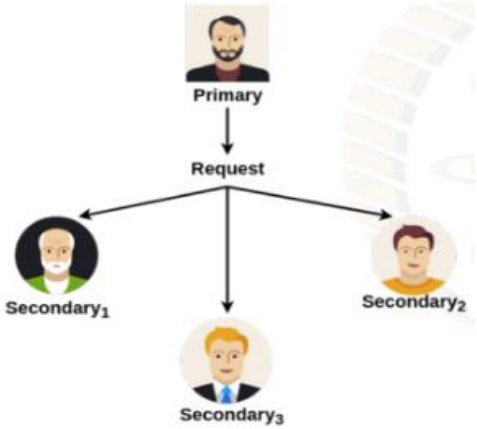
The practical byzantine fault tolerance has the following features,
• Asynchronous distributed system
– delay, out of order message
• Byzantine failure handling
– arbitrary node behavior
• Privacy
– tamper-proof message, authentication

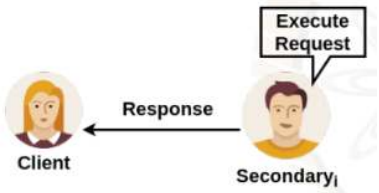
Simplified PBFT

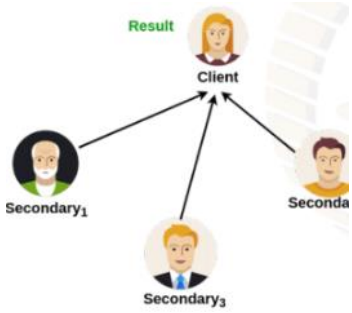
A state machine is replicated across different nodes
• 3f + 1 replicas are there where f is the number of faulty replicas
• The replicas move through a successions of configurations, known as views
• One replica in a view is primary and others are backups
• Views are changed when a primary is detected as faulty
• Every view is identified by a unique integer number ‘v’
• Only the messages from the current views are accepted
• Views are changed when a primary is detected as faulty
• Every view is identified by a unique integer number ‘v’
• Only the messages from the current views are accepted



Step: 1 Client request primary
A client sends a request to invoke a service operation to the primary.
Step: 2 Primary node broadcast the request to the backups.







Step: 3 Backups execute the request and send a reply to the client

The client waits for f + 1 replies from different backups with the same result

– f is the maximum number of faulty replicas that can be tolerated

Three Phase Commit Protocol

1) **Pre-Prepare**

Pre-prepare: Primary assigns a sequence number nn to the request and multicast a message << PRE-PREPARE, v, n, d >σ_p, m > to all the backups

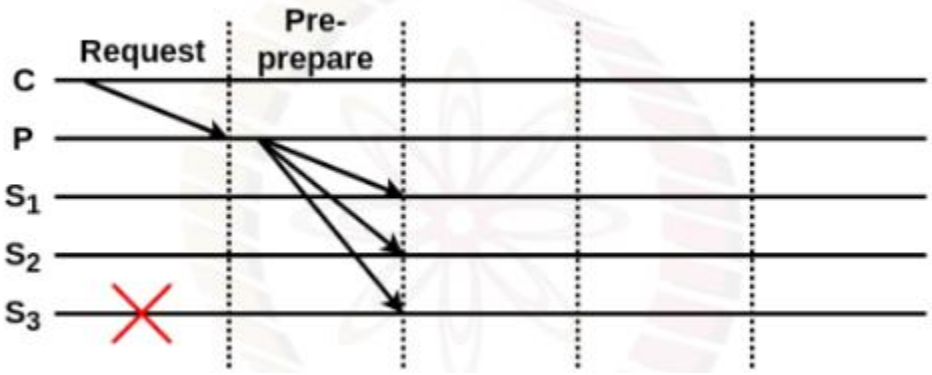
– v is the current view number

– n is the message sequence number

– d is the message digest

– σ_p is the private key of primary - works as a digital signature

– m is the message to transmit



– Acknowledge the request by a unique sequence number

Pre-prepare messages are used as a proof that request was assigned sequence number n is the view v

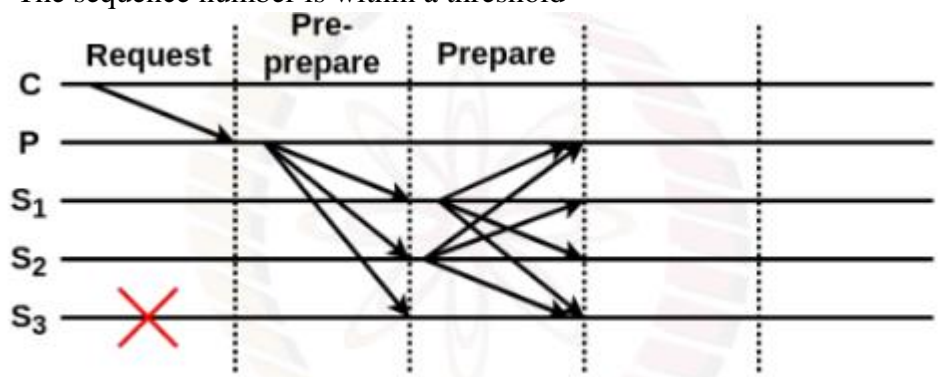
• A backup accepts a pre-prepare message if

– The signature is correct and d is the digest for mm

– The backup is in view v

– It has not received a different PRE-PREPARE message with sequence n and view v with a different digest

– The sequence number is within a threshold



2) **Prepare:**

– Replicas agree on the assigned sequence number

• If the backup accepts the PRE-PREPARE message, it enters prepare phase by multicasting a message < P, v, n, d, i >σ_ii to all other replicas

• A replica (both primary and backups) accepts prepare messages if

– Signatures are correct

– View number equals to the current view

– Sequence number is within a threshold

• Pre-prepare and prepare ensure that non-faulty replicas guarantee on a total order for the requests within a view

3) **Commit**

To commit a message if

– 2f prepares from different backups matches with the corresponding pre-prepare

– You have total 2f + 1 votes (one from primary that you already have!) from the non-faulty replicas

• Why do you require 3f + 1 replicas to ensure safety in an asynchronous system when there are f faulty nodes?

– If you have 2f+ 1 replicas, you need all the votes to decide the majority - boils down to a synchronous system

– You may not receive votes from certain replicas due to delay, in case of an asynchronous system

– f + 1 votes do not ensure majority, may be you have received f votes from Byzantine nodes, and just one vote from a non-faulty node (note Byzantine nodes can vote for or against - You do not know that a priori!)

– If you do not receive a vote

• The node is faulty and not forwarded a vote at all

• The node is non-faulty, forwarded a vote, but the vote got delayed

– Majority can be decided once 2f + 1 votes have arrived - even if f are faulty, you know f + 1 are from correct nodes, do not care about the remaining f votes

- Multicast < COMMIT, vv, nn, dd, ii >σσ_ii message to all the replicas including primary

• Commit a message when a replica

– Has sent a commit message itself
– Has received 2f + 1 commits (including its own)