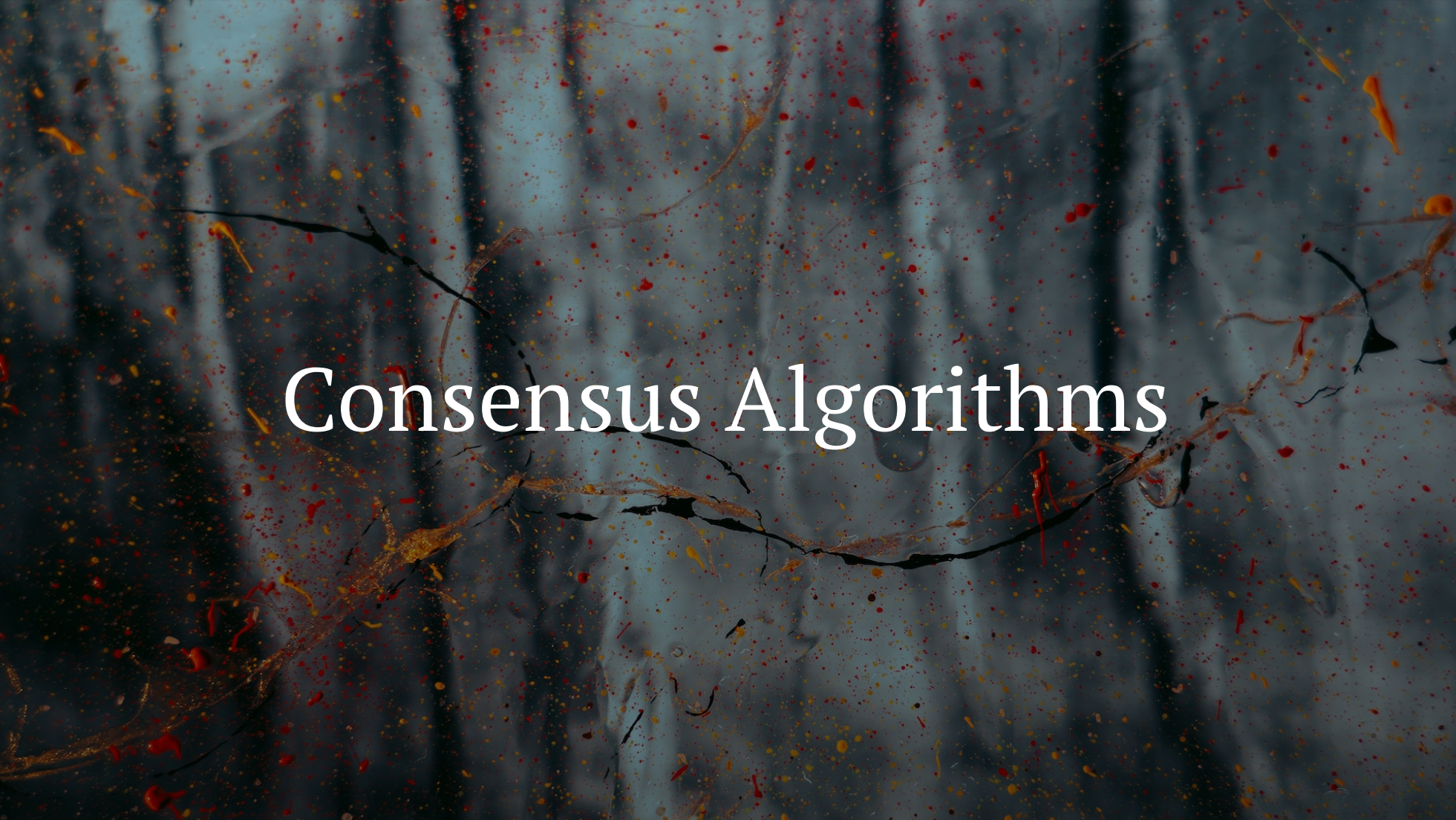

Consensus Algorithms

In this chapter we'll discuss:

- What is consensus, and why is it important
- Traditional consensus algorithms
- Byzantine fault tolerant algorithms
- Common consensus algorithms used in public blockchains such as Proof of Work (PoW), Proof of Stake (PoS) and other BFT Protocols
- How to chose a consensus algorithm

What is Consensus Anyways?

If you've read anything about how the blockchain works, you've almost definitely run across the word *"consensus"*. But what does it mean?

According to Wikipedia, consensus is a fundamental problem in distributed computing and multi-agent systems in order to achieve overall system reliability in the presence of a number of fault processes.

- This often requires coordinating processes to reach consensus, or agree on some data value that is needed during computation.

> In other words, it's the problem of getting a bunch of nodes in a network to agree on what's right and what's wrong. When you think about it, and how bad actors might exist, this problem is more difficult than it might initially seem.

Blockchain is a distributed system that relies upon a consensus mechanism, which ensures the safety and liveness of the blockchain network.

Introduction to Consensus Cont.

The distributed consensus problem has been studied extensively in distributed systems research since the late 1970s.

In the context of blockchain, we are concerned with the message passing type of distributed systems, where participants on the network communicate with each other via passing messages to each other.

In the past decade, the rapid evolution of blockchain technology has been observed. Also, with this tremendous growth, research regarding distributed consensus has grown significantly.

- A common research area is to convert traditional (classical) distributed consensus mechanisms into their blockchain variants that are suitable for blockchain networks.
- Another area of interest is to analyze existing and new consensus protocols.

> Spoiler: If you'd like to do research in this field, there's a lot of work to be done here

Introduction to Consensus Cont.

As mentioned back in the first couple chapters, there are different types of blockchain networks, particularly **permissioned** and **public** (non-permissioned). Obviously how we'll reach a consensus on a blockchain depends heavily on the type of blockchain.

- Permissioned blockchains are blockchain networks that require access to be a part of. Ex: Hyperledger.
- Public or non-permissioned blockchains allow anyone to participate and do transactions, and even participate in the consensus method. Ex: Bitcoin, Ethereum

For example, public blockchains employ Proof-of-Work (PoW) or Proof-of-Stake (PoS) for their consensus algorithm, but these algorithms might be pointless if employed in a permissioned blockchain.

Permissioned blockchains tend to run variants of traditional or classical distributed consensus, but the algorithms used here would be dangerous and impractical in public blockchains, as we won't be protected against bad actors and attacks.

So it's safe to say that these algorithms are *very* important.

The Byzantine Generals Problem

In distributed systems, a common goal is to achieve consensus (agreement) among nodes on the network even in the presence of faults. In order to explain the problem, an allegorical representation of the problem called the **Byzantine generals problem** was created that goes as such:

> *The Byzantine generals problem metaphorically depicts a situation where a Byzantine army, divided into different units, is spread around a city. A general commands each unit, and they can only communicate with each other using a messenger. To be successful, the generals must coordinate their plan and decide whether to attack or retreat.*
>
> *The problem, however, is that any generals could potentially be disloyal and act maliciously to obstruct agreement upon a united plan. The requirement now becomes that every honest general must somehow agree on the same decision even in the presence of treacherous generals.*
>
> *In order to address this issue, honest (loyal) generals must reach a majority agreement on their plan.*

Original Paper

(Fun fact: The person normally accredited with coming up with this problem is the same person behind LaTeX and has won a Turing award for his work on this problem (among other things))

In the digital world, generals are represented by computers (nodes) and communication links are messengers carrying messages. Disloyal generals are faulty nodes. The challenge here is to reach an agreement even in the presence of faults.

Fault Tolerance

A fundamental requirement in a consensus mechanism is that it must be **fault-tolerant**.

- In other words, it must be able to tolerate a number of failures in a network and should continue to work even in the presence of faults.
- This naturally means that there has to be some limit to the number of faults a network can handle, since no network can operate correctly if a large majority of its nodes are failing.

Types of Fault-Tolerant Consensus

Fault-tolerant algorithms can be divided into two types of fault- tolerance:

1. Crash fault-tolerance (CFT)
2. Byzantine fault-tolerance (BFT)

CFT covers only crash faults or, in other words, benign faults. In contrast, BFT deals with the type of faults that are arbitrary and can even be malicious.

We'll talk more about these algorithms in later slides.

Replication

> **Replication** is a standard approach to make a system fault-tolerant.

- Replication results in a synchronized copy of data across all nodes in a network.
- This technique improves the fault tolerance and availability of the network.
- This means that even if some of the nodes become faulty, the overall system/network remains available due to the data being available on multiple nodes.

There are two main types of replication techniques:

1. *Active* replication, which is a type where each replica becomes a copy of the original state machine replica.
2. *Passive* replication, which is a type where there is only a single copy of the state machine in the system kept by the primary node, and the rest of the nodes/replicas only maintain the state.

State Machine Replication

> A standard technique used to achieve fault tolerance in distributed systems, state machine replication (SMR) is a de facto technique that is used to prove deterministic replication services in order to achieve fault tolerance in a distributed system.

> A state machine at an abstract level, is a mathematical model that is used to describe a machine that can be in different states and can only be at only one state at a time.

- A state machine stores a state of the system and transitions it to the next state as a result of input received. As a result of state transition, an output is produced along with an updated state.

The idea behind SMR can be summarized as follows:

1. All servers always start with the same initial state.
2. All servers receive requests in a totally ordered fashion (sequenced as generated from clients).
3. All servers produce the same deterministic output for the same input.

State Machine Replication (Cont.)

- State machine replication is implemented under a primary/backup paradigm, where a primary node is responsible for receiving and broadcasting client requests.

> This broadcast mechanism is called **total order broadcast** or **atomic broadcast**, which ensures that backup or replica nodes receive and execute the same requests in the same sequence as the primary.

- Consequently, this means that all replicas will eventually have the same state as the primary, thus resulting in achieving consensus.
- In other words, this means that total order broadcast and distributed consensus are equivalent problems; if you solve one, the other is solved too.

Fault Tolerance in Distributed computing

One thing that's extremely important to note is that fault tolerance works up to a certain threshold.

- For example, if a network has a vast majority of constantly failing nodes and communication links, it is not hard to understand that this type of network may not be as fault-tolerant as we might like it to be.

In other words, even in the presence of fault- tolerant measures, if there is a lack of resources on a network, the network may still not be able to provide the required level of fault tolerance.

- In some scenarios, it might be impossible to provide the required services due to a lack of resources in a system.

> In distributed computing, such impossible scenarios are researched and reported as *impossibility results*.

Fault Tolerance in Distributed computing Cont.

In distributed computing, impossibility results provide an understanding of whether a problem is solvable and the minimum resources required to do so.

- If the problem is unsolvable, then these results give a clear understanding that a specific task cannot be accomplished and no further research is necessary.
- From another angle, we can say that impossibility results (sometimes called unsolvability results) show that certain problems are not computable under insufficient resources.
- Impossibility results unfold deep aspects of distributed computing and enable us to understand why certain problems are difficult to solve and under what conditions a previously unsolved problem might be solved.

Lower bound results

> The requirement of minimum available resources is known as **lower bound results**.

- We can think of lower bound as a minimum amount of resources, for example, the number of processors or communication links required to solve a problem.
- In other words, if a minimum required number of resources is not available in a system, then the problem cannot be solved.
- In the context of a consensus problem, a fundamental proven result is lower bounds on the number of processors

> The problems that are not solvable under any conditions are known as **unsolvability results**.

- For example, it has been proven that asynchronous deterministic consensus is impossible, also known as the **FLP impossibility result**.

FLP Impossibility

> *FLP impossibility* is a fundamental unsolvability result in distributed computing theory that states that in an asynchronous environment, the deterministic consensus is impossible, even if only one process is faulty.

- An **asynchronous environment** as it's name implies does not require all nodes to be synchronized. Real world communications are usually asynchronous
- By *determinstic*, it's obviously meant that we can predict the outcome of the algorithm every single time. In other words, there aren't any elements of randomness. We'll talk more about determinism in consensus algorithms near the end of this chapter.
- FLP is named after the authors' names, Fischer, Lynch and Patterson.

FLP Impossibility Cont.

To circumvent FLP impossibility, several techniques have been introduced in the literature. These techniques include:

- **Failure detectors**, which can be seen as oracles associated with processors to detect failures.

> Oracles in this context means an outside source of information we can trust. We'll see another kind of oracle later on when we want to bring outside information into our blockchain.

- **Randomized algorithms** have been introduced to provide a probabilistic termination guarantee. The core idea behind the randomized protocols is that the processors in such protocols can make a random choice of decision value if the processor does not receive the required quorum of trusted messages.

- **Synchrony assumptions**, where additional synchrony and timing assumptions are made to ensure that the consensus algorithm terminates and makes progress. We'll talk more about synchrony in later slides

Lower Bounds for Consensus

As we described previously, there are proven results in distributed computing that state several lower bounds, for example, the minimum number of processors required for consensus or the minimum number of rounds required to achieve consensus.

- The most common and fundamental of these results is the minimum number of processors required for consensus.

These results are:

- In the case of CFT, at least $2F + 1$ number of nodes is required to achieve consensus.
- In the case of BFT, at least $3F + 1$ number of nodes is required to achieve consensus.

F represents the number of failures.

Analysis and Design

In order to analyze and understand a consensus algorithm, we need to define a model under which our algorithm will run.

Model

Distributed computing systems represent different entities in the system under a computational model.

- This computational model is a beneficial way of describing the system under some system assumptions.

> A computational model represents processes, network conditions, timing assumptions, and how all these entities interact and work together.

Processes

Processes communicate with each other by passing messages to each other.

- This is why these systems are called message-passing distributed systems.
- There is another class, called shared memory, which we will not discuss here as we are only dealing with message-passing systems.

Timing Assumptions

Some assumptions in regards to timing are made when designing consensus algorithms:

- **Synchrony**: In synchronous systems, there is a known upper bound on the communication and processor delays.
  - Synchronous algorithms are designed to be run on synchronous networks.
  - At a fundamental level, in a synchronous system, a message sent by a processor to another is received by the receiver in the same communication round as it is sent.

- **Asynchrony**: In asynchronous systems, there is no upper bound on the communication and processor delays.
  - In other words, it is impossible to define an upper bound for communication and processor delays in asynchronous systems.
  - Asynchronous algorithms are designed to run on asynchronous networks without any timing assumptions.
  - These systems are characterized by the unpredictability of message transfer (communication) delays and processing delays.
  - This scenario is common in large-scale geographically dispersed distributed systems and systems where the input load is unpredictable.

Timing Assumptions Cont.

- **Partial Synchrony**: In this model, there is an upper bound on the communication and processor delays, however, this upper bound is not known to the processors.
    - Eventually, synchronous systems are a type of partial synchrony, which means that the system becomes synchronous after an instance of time called **global stabilization time or GST**.
    - GST is not known to the processors.
    - Generally, partial synchrony captures the fact that,usually, the systems are synchronous, but there are arbitrary but bounded asynchronous periods.
    - Also, the system at some point is synchronous for long enough that processors can decide (achieve agreement) and terminate during that period.

Consensus Algorithm Classification

There are two main classes of consensus algorithms. Generally, they are:

- **Traditional—voting-based consensus**, also known as *fault-tolerant distributed consensus*. Ex: Paxos, PBFT
- **Lottery-based—Nakamoto** (also known as *blockchain consensus*) and **post-Nakamoto consensus**. Ex: PoW, PoS
- The fundamental requirements of consensus algorithms boil down to *safety* and *liveness* conditions.
- A consensus algorithm must be able to satisfy the safety and liveness properties.

> Safety is usually based on some safety requirements of the algorithms, such as agreement, validity, and integrity.

> Liveness means that the protocol can make progress even if the network conditions are not ideal.

Safety and Liveness

Safety

- This requirement generally means that nothing bad happens.
- There are usually three properties within this class of requirements, which are listed as follows:
    - **Agreement**: The agreement property requires that no two processes decide on different values.
    - **Validity**: Validity states that if a process has decided a value, that value must have been proposed by a process. In other words, the decided value is always proposed by an honest process and has not been created out of thin air.
    - **Integrity**: A process must decide only once.

Liveness

- This requirement generally means that something good eventually happens.
    - **Termination**: This liveness property states that each honest node must eventually decide on a value.

Crash Fault Tolerance (CFT) Algorithms: Paxos

The most fundamental distributed consensus algorithm, it allows consensus over a value under unreliable communications.

- In other words, Paxos is used to build a reliable system that works correctly, even in the presence of faults.
- Paxos makes use of $2F + 1$ processes to ensure fault tolerance in a network where processes can crash fault, that is, experience benign failures.
    - In other words, Paxos can tolerate one crash failure in a three-node network.

> Benign failure means either the loss of a message or a process stops.

Paxos is a two-phase protocol.

- The first phase is called the *prepare* phase, and the next phase is called the *accept* phase.
- Paxos has *proposers* and *acceptors* as participants, where the proposer is the replicas or nodes that propose the values and acceptors are the nodes that accept the value.

How Paxos Works

The Paxos protocol assumes an asynchronous message-passing network with less than 50% of crash faults.

- As usual, the critical properties of the Paxos consensus algorithm are *safety* and *liveness*.

Under safety, we have:

- **Agreement**, which specifies that no two different values are agreed on. In other words, no two different learners learn different values.
- **Validity**, which means that only the proposed values are decided. In other words, the values chosen or learned must have been proposed by a processor.

Under liveness, we have:

- **Termination**, which means that, eventually, the protocol is able to decide and terminate. In other words, if a value has been chosen, then eventually learners will learn it.

Processes in Paxos

Processes can assume different roles:

- **Proposers**, elected leader(s) that can propose a new value to be decided.
- **Acceptors**, which participate in the protocol as a means to provide a majority decision.
- **Learners**, which are nodes that just observe the decision process and value.

It should be noted that a single process in a Paxos network can assume all three roles.

- The key idea behind Paxos is that the proposer node proposes a value, which is considered final only if a majority of the acceptor nodes accept it. The learner nodes also learn this final decision.

How it works

More simply though, Paxos can be seen as a protocol that is quite similar to a simpler protocol known as the two-phase commit protocol.

> Two-phase commit (2PC) is a standard atomic commitment protocol to ensure that transactions are committed in distributed databases only if all participants agree to commit. Even if a single node cannot agree to commit the transaction, it is fully rolled back.

- Similarly, in Paxos, in the first phase, the proposer sends a proposal to the acceptors, if and when they accept the proposal, the proposer broadcasts a request to commit to the acceptors.
- Once the acceptors commit and report back to the proposer, the proposal is considered final, and the protocol concludes.
- In contrast with the two-phase commit, Paxos introduced ordering (sequencing to achieve total order) of the proposals and majority-based acceptance of the proposals instead of expecting all nodes to agree (to allow progress even if some nodes fail).
    - Both of these improvements contribute toward ensuring the safety and liveness of the Paxos algorithm.

How it works: Step-by-Step

1. The proposer proposes a value by broadcasting a message, `<prepare(n)>`, to all acceptors.

2. Acceptors respond with an acknowledgment message if proposal `n` is the highest that the acceptor has responded to so far.
    - The acknowledgment message `<ack(n, v, s)>` consists of three variables where `n` is the proposal number, `v` is the proposal value of the highest numbered proposal the acceptor has accepted so far, and `s` is the sequence number of the highest proposal accepted by the acceptor so far.
    - This is where acceptors agree to commit the proposed value.
    - The proposer now waits to receive acknowledgment messages from the majority of the acceptors indicating the **chosen** value.

3. If the majority is received, the proposer sends out the "accept" message `<accept(n, v)>` to the acceptors.

How Paxos Works Cont.

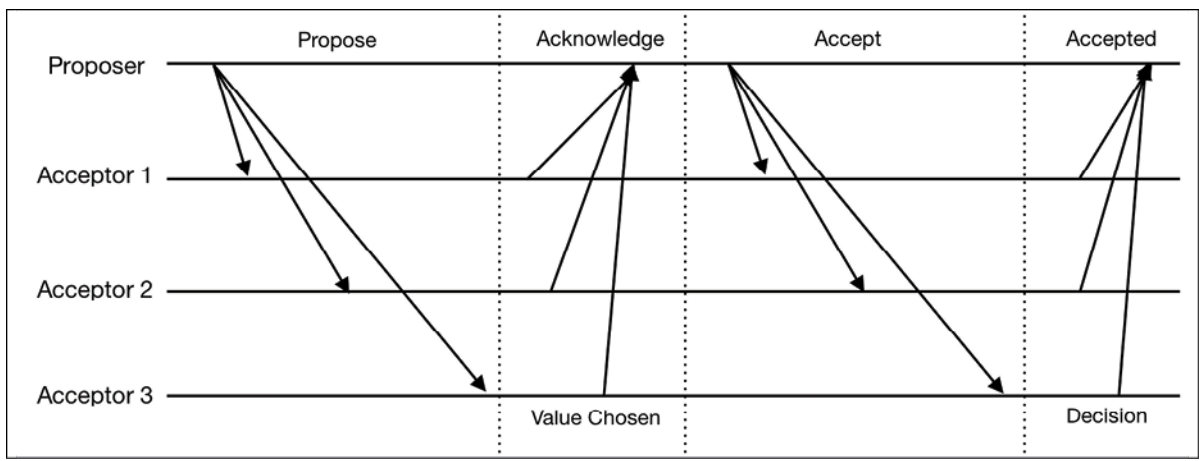
4. If the majority of the acceptors accept the proposed value (now the "accept" message), then it is decided: that is, agreement is achieved.

5. Finally, in the learning phase, acceptors broadcast the "accepted" message `<accepted(n, v)>` to the proposer.
    - This phase is necessary to disseminate which proposal has been finally accepted.
    - The proposer then informs all other learners of the decided value.
    - Alternatively, learners can learn the decided value via a message that contains the accepted value (decision value) multicast by acceptors.

Paxos summary

In summary, the key points to remember about Paxos are that:

- First, a proposer suggests a value with the aim that acceptors achieve agreement on it.
- The decided value is a result of majority consensus among the acceptors and is finally learned by the learners.



How Paxos Achieves Safety and liveness

We've talked about how Paxos has safety and liveness, but how exactly does this algorithm provide these two attributes despite being so simple?

The actual proofs for the correctness are beyond the scope of this class, but the intuition is as follows:

- **Agreement** is ensured by enforcing that only one proposal can win votes from a majority of the acceptors.
- **Validity** is ensured by enforcing that only the genuine proposals are decided. In other words, no value is committed unless it is proposed in the proposal message first.
- **Liveness** or termination is guaranteed by ensuring that at some point during the protocol execution, eventually there is a period during which there is only one fault-free proposer.

Raft

Another easy-to-understand CFT consensus mechanism where the leader is always assumed to be honest.

At a conceptual level, it is a replicated log for a replicated state machine (RSM) where a unique leader is elected every "term" (time division) whose log is replicated to all follower nodes.

Raft is composed of three sub-problems:

1. **Leader election** (a new leader election in case the existing one fails)
2. **Log replication** (leader to follower log synch)
3. **Safety** (no conflicting log entries (index) between servers)

- Each server in Raft can have either a **follower**, **leader**, or **candidate** state.
- The protocol ensures election **safety** (that is, only one winner each election term) and **liveness** (that is, some candidate must eventually win).

How Raft Works

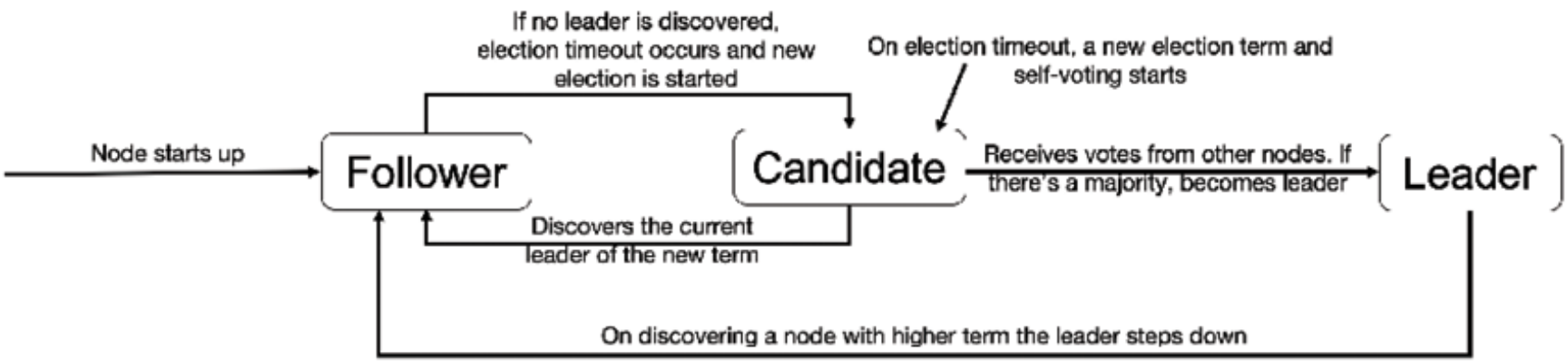
At a fundamental level, the protocol is quite simple and can be described simply by the following sequence:

Node starts up –> Leader election –> Log replication

1. First, the node starts up.

2. After this, the leader election process starts. Once a node is elected as leader, all changes go through that leader.

3. Each change is entered into the node's log.

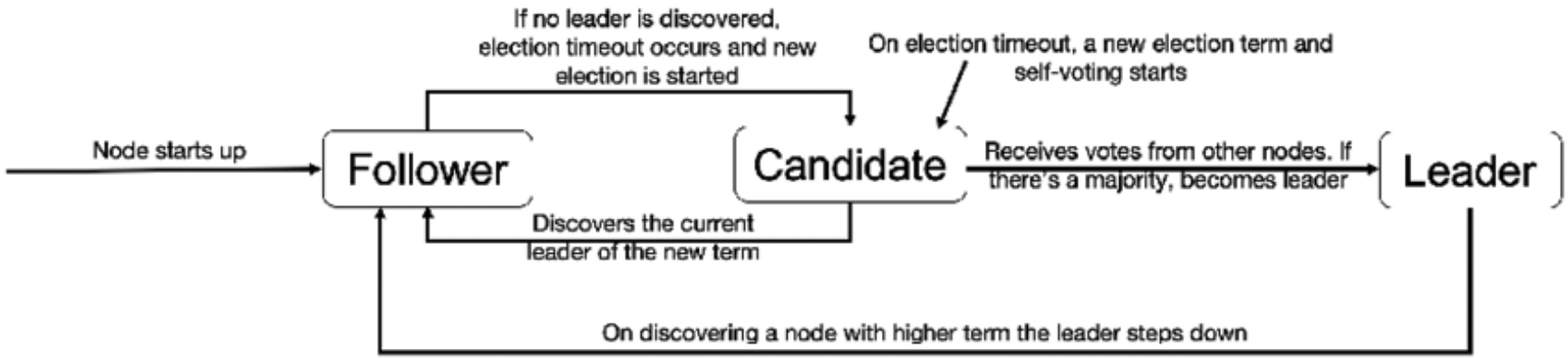


How Raft Works Cont.

4. Log entry remains uncommitted until the entry is replicated to follower nodes and the leader receives write confirmation votes from a majority of the nodes, then it is committed locally.

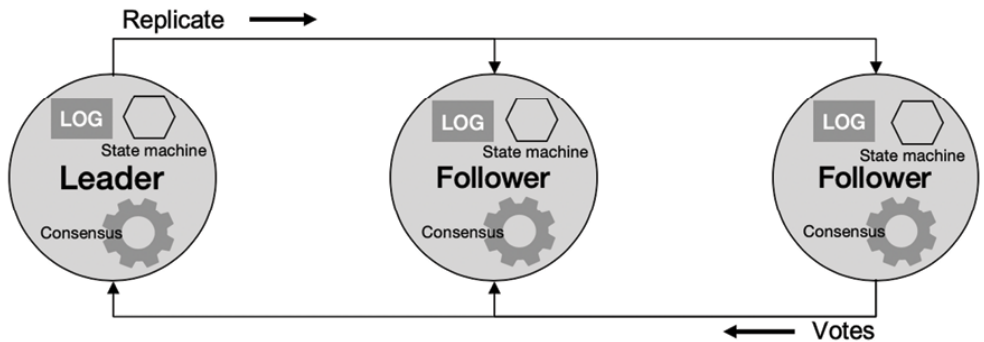
5. The leader notifies the followers regarding the committed entry.

6. Once this process ends, agreement is achieved.



Log Replication

- In Raft, the log (data) is eventually replicated across all nodes.
    - The aim of log replication is to synchronize nodes with each other.



- As shown in the preceding diagram, the leader is responsible for log replication.
- Once the leader has a new entry in its log, it sends out the requests to replicate to the follower nodes.
- When the leader receives enough confirmation votes back from the follower nodes indicating that the replicate request has been accepted and processed by the followers, the leader commits that entry to its local state machine.
    - At this stage, the entry is considered committed.

BFT Algorithms: Practical Byzantine Fault Tolerance (PBFT)

> *Reminder*: Simply put, BFT algorithms are algorithms where we might have bad actors in our network on top of pre-existing error-inducing factors such as crashes and network loss.

PBFT, as the name suggests, is a protocol developed to provide consensus in the presence of Byzantine faults.

- Before PBFT, Byzantine fault tolerance was considered impractical.
- With PBFT, it was demonstrated for the first time that practical Byzantine fault tolerance is possible.
- PBFT is based on state-machine replication, mentioned briefly in earlier slides.

> **State machine replication** is a mechanism that allows synchronization between replicas/nodes of the network with every node agreeing on a single "state" at any given time.

PBFT Sub-Protocols

PBFT comprises three sub-protocols:

- **Normal operation**: Normal operation sub-protocol refers to a scheme that is executed when everything is running normally and no errors are in the system.
- **View change**: View change is a sub-protocol that runs when a faulty leader node is detected in the system.
- **Checkpointing**: Checkpointing is another sub- protocol, which is used to discard the old data from the system.

PBFT Overview

The PBFT protocol comprises three phases or steps. These phases run in a sequence to achieve consensus.

- The protocol runs in rounds where, in each round, an elected leader node, called the *primary node*, handles the communication with the client.
- In each round, the protocol progresses through the three phases: **pre-prepare**, **prepare**, and **commit**. These phases are run in a sequence to achieve consensus.
- The participants in the PBFT protocol are called *replicas*, where one of the replicas becomes primary as a leader in each round, and the rest of the nodes acts as backups.
    - Here, each node maintains a local log, and the logs are kept in sync with each other via the consensus protocol: that is, PBFT.

As we saw earlier, in order to tolerate Byzantine faults, the minimum number of nodes required is $N = 3F + 1$, where N is the number of nodes and F is the number of faulty nodes.

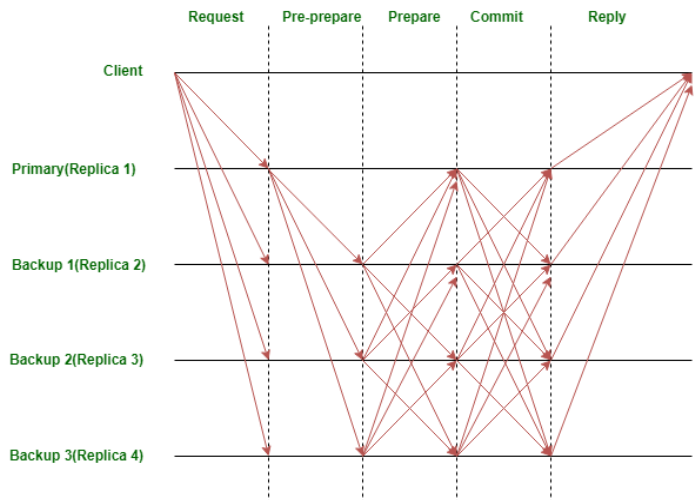
- PBFT ensures Byzantine fault tolerance as long as the number of nodes in a system stays $N >= 3F + 1$.

How PBFT Works (Simply)

> In summary, when a client sends a request to a primary (the elected leader node), the protocol initiates a sequence of operations between replicas (participants), which eventually leads to consensus and a reply back to the client.

Less simply:

1. The client sends a request to the primary (leader) node.
2. The primary (leader) node broadcasts the request to the all the secondary (backup) nodes.
3. The nodes(primary and secondaries) perform the service requested and then send back a reply to the client.
4. The request is served successfully when the client receives $m + 1$ replies from different nodes in the network with the same result, where m is the maximum number of faulty nodes allowed.



How PBFT Works (Even Less Simply)

This algorithm is a form of state machine replication.

- In other words, the service is modeled as a state machine that is replicated across different nodes in a distributed system.

The sequence of operations are divided into different phases:

- Pre-prepare
- Prepare
- Commit

How PBFT Works (Even Less Simply) Cont.

In addition, each replica maintains a local state comprising three main elements:

1. Service state
   - By service, the "service" the network of nodes aims to provide. This *service* could be a distributed ledger, such as a blockchain, for example.
   - At any given moment, this service could be in a specific state. This varies from application to application, but the best way to think of what a state could be is to think of a turing machine distributed over a large number of computers/processors/nodes that can be in a certain state at any given moment.
   - Additionally, this service has *operations*, that using these operations, the service's state could change.
   - Clients could issue requests to the replicated service to invoke operations to receive a reply after a certain amount of time.

2. A message log

3. A number representing that replica's current view
   - A successful configuration of our state machine is called a *view* here.
   - In a view, one replica is a primary (leader) and the other replicas are called backups.
   - Views are numbered consecutively such that in view n, the n^{th} replica is the primary.
   - The view changes are carried out when it appears the primary has failed

Phase 1: Pre-prepare

- This is the first phase in the protocol, where the primary node, or primary, receives a request from the client.
- The primary node assigns a sequence number to the request.
- It then sends the pre- prepare message with the request to all backup replicas.

When the pre-prepare message is received by the backup replicas, it checks a number of things to ensure the validity of the message:

- First, whether the digital signature is valid.
- After this, whether the current view number is valid.
- Then, that the sequence number of the operation's request message is valid.
- Finally, if the digest/hash of the operation's request message is valid.

If all of these elements are valid, then the backup replica accepts the message. After accepting the message, it updates its local state and progresses toward the prepare phase.

More abstractly, the pre-prepare sub-protocol algorithm goes as such:

1. Accepts a request from the client.
2. Assigns the next sequence number.
3. Sends the pre-prepare message to all backup replicas.

Phase 2: Prepare

- A prepare message is sent by each backup to all other replicas in the system.
- Each backup waits for at least $2F + 1$ prepare messages to be received from other replicas.
- They also check whether the prepare message contains the same view number, sequence number, and message digest values.
- If all these checks pass, then the replica updates its local state and progresses toward the commit phase.

More abstractly, the prepare sub-protocol algorithm goes as such:

1. Accepts the pre-prepare message. If the backup has not accepted any pre-prepare messages for the same view or sequence number, then it accepts the message.
2. Sends the prepare message to all replicas.

Phase 3: Commit

- In the commit phase, each replica sends a commit message to all other replicas in the network.
- The same as the prepare phase, replicas wait for $2F + 1$ commit messages to arrive from other replicas.
  - The replicas also check the view number, sequence number, and message digest values.
- If they are valid for $2F + 1$ commit messages received from other replicas, then the replica executes the request, produces a result, and finally, updates its state to reflect a commit.
  - If there are already some messages queued up, the replica will execute those requests first before processing the latest sequence numbers.
- Finally, the replica sends the result to the client in a reply message.
- The client accepts the result only after receiving $2F + 1$ reply messages containing the same result.

More abstractly, the prepare sub-protocol algorithm goes as such:

1. The replica waits for 2F prepare messages with the same view, sequence, and request.
2. Sends a commit message to all replicas.
3. Waits until a 2F + 1 valid commit message arrives and is accepted.
4. Executes the received request.
5. Sends a reply containing the execution result to the client.

Certificates in PBFT

During the execution of the protocol, the integrity of the messages and protocol operations must be maintained to provide an adequate level of security and assurance. This is maintained by the use of digital signatures. In addition, certificates are used to ensure the adequate majority of participants (nodes).

- Note that these certificates are not usual digital certificates commonly used in PKI and IT infrastructures to secure assets such as servers.

Certificates in PBFT protocols are used to demonstrate that at least $2F + 1$ nodes have stored the required information.

- In other words, the collection of $2F + 1$ messages of a particular type is considered a certificate.

For example, if a node has collected $2F + 1$ messages of type prepare, then combining it with the corresponding pre-prepare message with the same view, sequence, and request represents a **certificate**, called a *prepared certificate*. Similarly, a collection of $2F + 1$ commit messages is called a *commit certificate*.

Extra PBFT Variables

There are also a number of variables that the PBFT protocol maintains in order to execute the algorithm:

State Variable	Explanation
v	View number
m	Last request message
n	Sequence number of the message
h	Hash of the message
i	Index number
C	Set of all checkpoints
P	Set of all pre-prepare and corresponding prepare messages
O	Set of pre-prepare messages without corresponding request messages

Types of Messages

Message	From	To	Format	Signed By
Request	Client	Primary	`<REQUEST, m>`	Client
Pre-Prepare	Primary	Backups	`<PRE-PREPARE, v, n, h>`	Client
Prepare	Replica	Backups	`<PREPARE, v, n, h, i>`	Replica
Commit	Replica	Replicas	`<COMMIT, v, n, h, i>`	Replica
Reply	Replicas	Client	`<REPLY, r, i>`	Replica
View Change	Replica	Replicas	`<VIEWCHANGE, v+1, n, C, P, i>`	Replica
New View	Primary replica	Replicas	`<NEWVIEW, v + 1, v, O>`	Replica
Checkpoint	Replica	Replicas	`<CHECKPOINT, n, h, i>`	Replica

Specific message types

`<VIEWCHANGE, v+1, n, C, P, i>`

> View-change occurs when a primary is suspected faulty.

This phase is required to ensure protocol progress. With the view change sub-protocol, a new primary is selected, which then starts normal mode operation again.

- The new primary is selected in a round-robin fashion.

When a backup replica receives a request, it tries to execute it after validating the message, but for some reason, if it does not execute it for a while, the replica times out and initiates the view change sub- protocol.

In the view change protocol, the replica stops accepting messages related to the current view and updates its state to view-change. The only messages it can receive in this state are checkpoint messages,view-change messages, and new-view messages. After that, it sends a view-change message with the next view number to all replicas.

View-Change Cont.

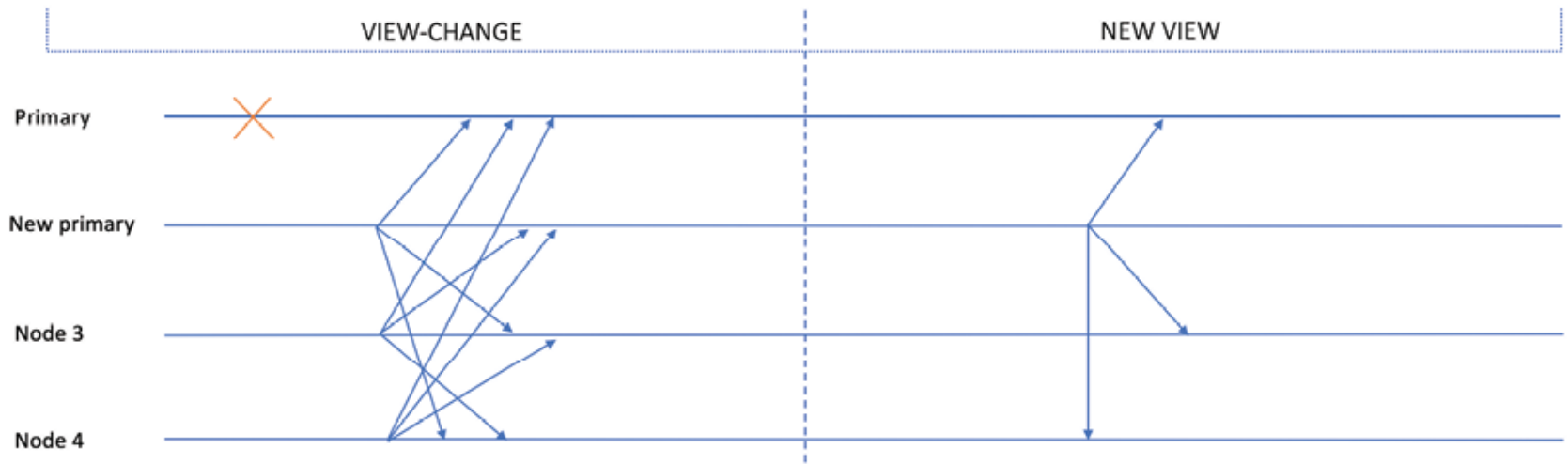
- When this message arrives at the new primary, the primary waits for at least $2F$ view-change messages for the next view.
- If at least 2F view-change messages are received it broadcasts a new view message to all replicas and progresses toward running normal operation mode once again.
- When other replicas receive a new-view message, they update their local state accordingly and start normal operation mode.

The algorithm for the view-change protocol is shown as follows:

1. Stop accepting pre-prepare, prepare, and commit messages for the current view.
2. Create a set of all the certificates prepared so far.
3. Broadcast a view-change message with the next view number and a set of all the prepared certificates to all replicas.

View-Change Cont.

The view-change protocol can be visualized through this diagram:



Why this sub-protocol is important

The view-change sub-protocol is a mechanism to achieve **liveness**.

Three smart techniques are used in this sub-protocol to ensure that, eventually, there is a time when the requested operation executes:

1. A replica that has broadcast the view-change message waits for $2F + 1$ view-change messages and then starts its timer.
    - If the timer expires before the node receives a new-view message for the next view, the node will start the view change for the next sequence but will increase its timeout value.
    - This will also occur if the replica times out before executing the new unique request in the new view.

2. As soon as the replica receives $F + 1$ view-change messages for a view number greater than its current view, the replica will send the view-change message for the smallest view it knows of in the set so that the next view change does not occur too late.
    - This is also the case even if the timer has not expired; it will still send the view change for the smallest view.

3. As the view change will only occur if at least $F + 1$ replicas have sent the view-change message, this mechanism ensures that a faulty primary cannot indefinitely stop progress by successively requesting view changes.

Checkpoint Sub-Protocol

> This sub-protocol could be thought of replica housekeeping

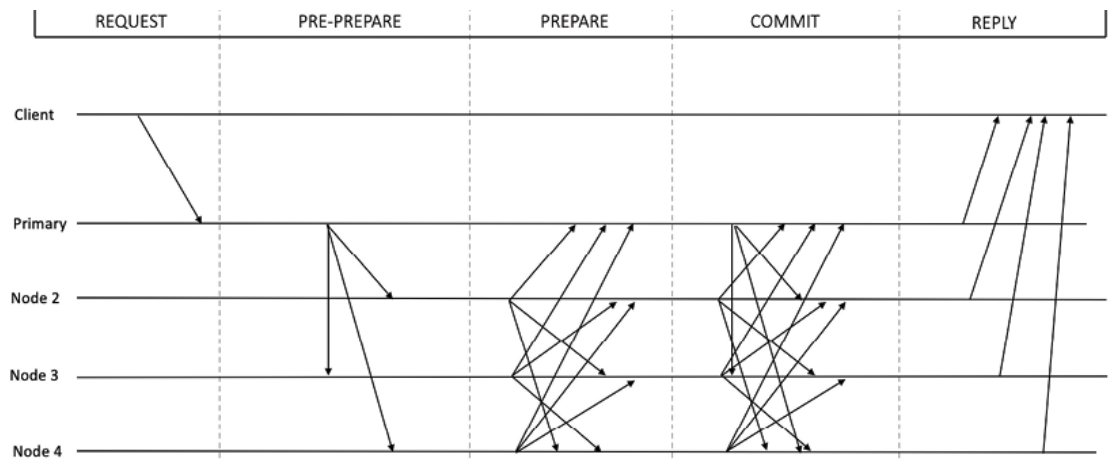
Checkpointing is a crucial sub-protocol. It is used to discard old messages in the log of all replicas.

- With this, the replicas agree on a stable checkpoint that provides a snapshot of the global state at a certain point in time.
- This is a periodic process carried out by each replica after executing the request and marking that as a checkpointing its log.
- A variable called *low watermark* (in PBFT terminology) is used to record the sequence number of the last stable checkpoint. This checkpoint is then broadcast to other nodes.
- As soon as a replica has at least $2F + 1$ checkpoint messages, it saves these messages as proof of a stable checkpoint.

It discards all previous pre-prepare, prepare, and commit messages from its logs.

PBFT In Summary

In summary, the primary purpose of these phases is to achieve consensus, where each phase is responsible for a critical part of the consensus mechanism, which after passing through all phases, eventually ends up achieving consensus.



You can read PBFT's paper here. *(It's an easy read, I promise)*

PBFT Strengths & Weaknesses

Strengths:

- PBFT provides immediate and deterministic transaction finality.
  - This is in contrast with the PoW protocol, where a number of confirmations are required to finalize a transaction with high probability.
- PBFT is also energy efficient as compared to PoW, which consumes a tremendous amount of electricity.

Weaknesses:

- PBFT is not very scalable.
  - This is the reason it is more suitable for consortium networks, instead of public blockchains.
  - It is, however, much faster than PoW protocols.

- Sybil attacks can be carried out on a PBFT network, where a single entity can control many identities to influence the voting and subsequently the decision.
  - However, the fix is trivial and, in fact, this is not very practical in consortium networks where all identities are known on the network.
  - This problem can be addressed simply by increasing the number of nodes in the network.

Istanbul Byzantine Fault Tolerance

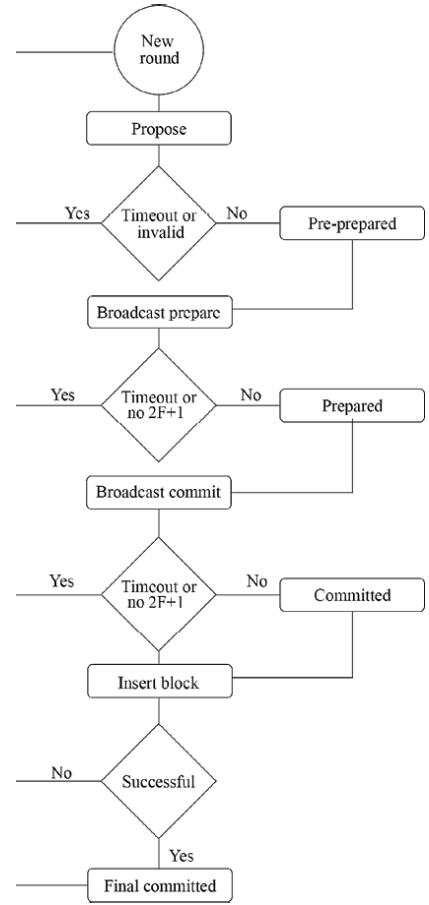
> IBFT was developed by AMIS Technologies as a variant of PBFT suitable for blockchain networks. However, there are some obvious differences between IBFT AND PBFT.

- There is no distinctive concept of a client in IBFT.
  - Instead, the proposer can be seen as a client, and in fact, all validators can be considered clients.
- There is a concept of dynamic validators, which is in contrast with the original PBFT, where the nodes are static.
  - However, in IBFT, the validators can be voted in and out as required.
- There are two types of nodes in an IBFT network, **nodes** and **validators**.
  - Nodes are synchronized with the blockchain without participating in the IBFT consensus process.
  - In contrast, validators are the nodes that participate in the IBFT consensus process.
- IBFT relies on a more straightforward structure of view-change (round change) messages as compared to PBFT.
- In contrast with PBFT, in IBFT there is no concrete concept of checkpoints.
  - However, each block can be considered an indicator of the progress so far (the chain height).
- There is no concept of garbage collection in IBFT.

How IBFT Works

IBFT assumes a network model under which it is supposed to run;

- The model is composed of at least $3F + 1$ processes (standard BFT assumption), a partially synchronous message-passing network, and sound cryptography.
- By sound cryptography, it is assumed that digital signatures and relevant cryptographic protocols such as cryptographic hash functions are secure.
- The IBFT protocol runs in rounds.
    - It has three phases: **pre-prepare**, **prepare**, and **commit**.
    - In each round, usually, a new leader is elected based on a round-robin mechanism.



How IBFT Works

1. The protocol starts with a new round.
    - In the new round, the selected proposer broadcasts a proposal (block) as a pre-prepare message.

2. The nodes that receive this pre-prepare message validate the message and accept it if it is a valid message.
    - The nodes also then set their state to pre-prepared.

3. At this stage, if a timeout occurs, or a proposal is seen as invalid by the nodes, they will initiate a round change.
    - The normal process then begins again with a proposer, proposing a block.

4. Nodes then broadcast the prepare message and wait for $2F + 1$ prepare messages to be received from other nodes.
    - If the nodes do not receive 2F+1 messages in time, then they time out, and the round change process starts.
    - The nodes then set their state to prepared after receiving 2F+1 messages from other nodes.

How IBFT Works Cont.

5. Finally, the nodes broadcast a commit message and wait for $2F + 1$ messages to arrive from other nodes.
    - If they are received, then the state is set to committed, otherwise, timeout occurs and the round change process starts.

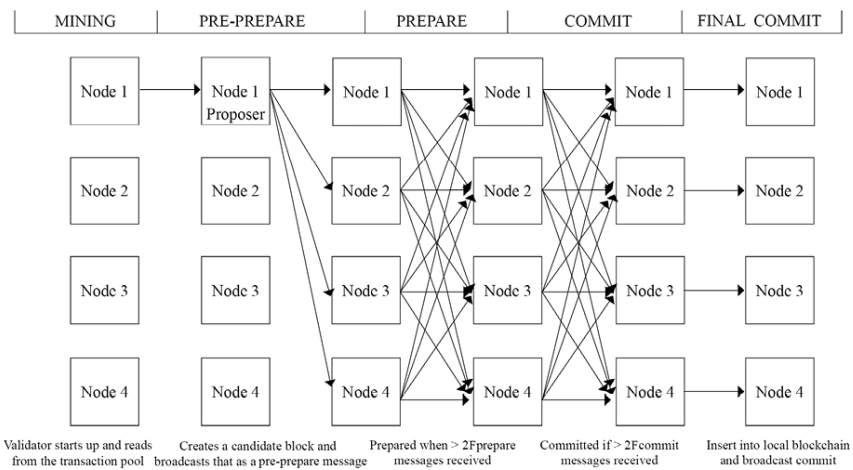
6. Once committed, block insertion is tried.
    - If it succeeds, the protocol proceeds to the final committed state and, eventually, a new round starts.
    - If insertion fails for some reason, the round change process triggers.
    - Again, nodes wait for 2F+1 round change messages, and if the threshold of the messages is received, then round change occurs.

IBFT States

There are a lot of mentions of "states" in IBFT. That's because IBFT is an SMR (state machine replication, introduced in previous slides) algorithm.

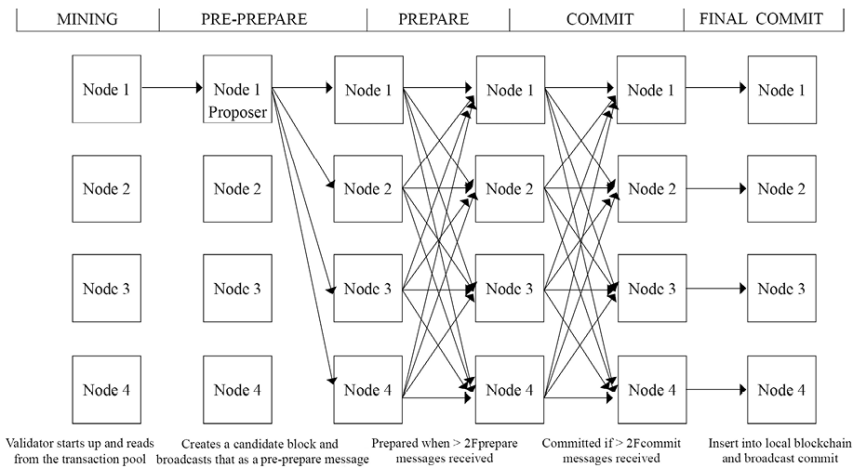
Each validator maintains a state machine replica in order to reach block consensus, that is, agreement. These states are:

- **New round**: In this state, a new round of the consensus mechanism starts, and the selected proposer sends a new block proposal to other validators. In this state, all other validators wait for the `PRE-PREPARE` message.
- **Pre-prepared**: A validator transitions to this state when it has received a `PRE-PREPARE` message and broadcasts a `PREPARE` message to other validators. The validator then waits for 2F + 1 `PREPARE` or `COMMIT` messages.



IBFT States Cont.

- **Prepared**: This state is achieved by a validator when it has received 2F+1 prepare messages and has broadcast the commit messages. The validator then awaits 2F+1 commit messages to arrive from other validators.
- **Committed**: The state indicates that a validator has received 2F+1 `COMMIT` messages. The validator at this stage can insert the proposed block into the blockchain.
- **Final committed**: This state is achieved by a validator when the newly committed block is inserted successfully into the blockchain. At this state, the validator is also ready for the next round of consensus.
- **Round change**: This state indicates that the validators are waiting for 2F+1 round change messages to arrive for the newly proposed new round number.



IBFT Validator Management

An additional mechanism that makes IBFT quite appealing is its validator management mechanism.

- By using this mechanism, validators can be added or removed by voting between members of the network.
- This is quite a useful feature and provides the right level of flexibility when it comes to managing validators efficiently, instead of manually adding or removing validators from the validator set.

> IBFT has beeen implemented in several blockchains, like Quorum and Celo

Tendermint Consensus Algorithm

Traditionally, a consensus mechanism used to run with a small number of participants and thus performance and scalability was not a big concern.

However, with the advent of blockchain, there is a need to develop algorithms that can work on wide area networks and in asynchronous environments.

Research into these areas of distributed computing is not new and especially now, due to the rise of cryptocurrencies and blockchain, the interest in these research topics has grown significantly in the last few years.

> Inspired by PBFT, Tenderment also makes use of the SMR approach to providing consensus.

How Tendermint Works

The Tendermint protocol also works by running rounds.

- In each round, a leader is elected, which proposes the next block.
  - Also note that in Tendermint, the round change or view-change process is part of the normal operation, as opposed to PBFT, where view-change only occurs in the event of errors, that is, a suspected faulty leader.

- Tendermint works similarly to PBFT, where three phases are required to achieve a decision.
  - Once a round is complete, a new round starts with three phases and terminates when a decision is reached.

- A key innovation in Tendermint is the design of a new termination mechanism.
  - As opposed to other PBFT-like protocols, Tendermint has developed a more straightforward mechanism, which is similar to PBFT-style normal operation.
  - Instead of having two sub-protocols for normal mode and view-change mode (recovery in event of errors), Tendermint terminates without any additional communication costs.

Tendermint System model

In previous slides we mentioned how each consensus model is studied and developed under a system model with some assumptions about the system.

Tendermint's system model is as follows;

- **Processes**: A process is the fundamental key participant of the protocol.
  - It is also called a replica (in PBFT traditional literature), a node, or merely a process.
  - Processes can be correct or honest. Processes can also be faulty or Byzantine.
  - Each process possesses some voting power. Also, note that processes are not necessarily connected directly; they are only required to connect loosely or just with their immediate subset of processes/nodes.
  - Processes have a local timer that they use to measure timeout.

- **Network model**: The network model is a network of processes that communicate using messages.
  - In other words, the network model is a set of processes that communicate using message passing.
  - Particularly, the gossip protocol is used for communication between processes.
  - The standard assumption of N >= 3F +1 BFT is also taken into consideration.

Tendermint System Model Cont.

- **Timing assumptions**: Under the network model, Tendermint assumes a partially synchronous network.
  - This means that there is an unknown bound on the communication delay (GST).

- **Security and cryptography**: It is assumed that the public key cryptography used in the system is secure and the impersonation or spoofing of accounts/identities is not possible.
  - The messages on the network are authenticated and verified via digital signatures.
  - The protocol ignores any messages with an invalid digital signature.

- **State machine replication**: To achieve replication among the nodes, the standard SMR mechanism is used.
  - One key observation that is fundamental to the protocol is that in SMR, it is ensured that all replicas on the network receive and process the same sequence of requests.
  - As noted in the Tendermint paper, agreement and order are two properties that ensure that all requests are received by replicas and order ensures that the sequence in which the replicas have received requests is the same.
  - Both of these requirements ensure total order in the system.
  - Also, Tendermint ensures that requests themselves are valid and have been proposed by the clients. In other words, only valid transactions are accepted and executed on the network.

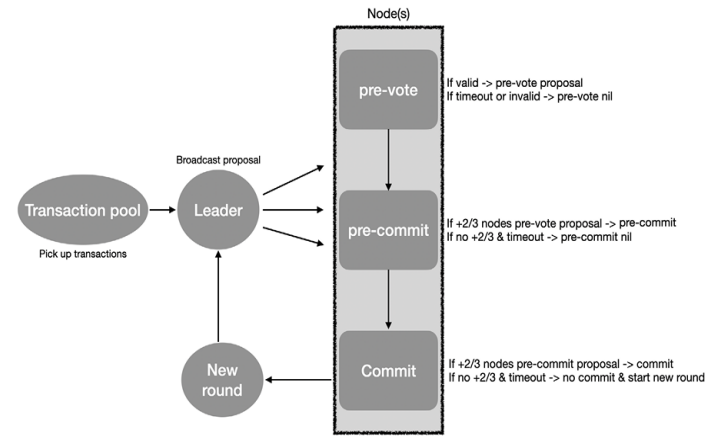
Safety and Liveness in Tendermint

Similar to consensus algorithms we've previously studied, in Tendermint, safety and liveness consists of *agreement*, *termination* and *validity*.

- **Agreement**: No two correct processes decide on different values.
- **Termination**: All correct processes eventually decide on a value.
- **Validity**: A decided upon value is valid, that is, it satisfies the predefined predicate denoted `valid()`.

How Tendermint Works In Abstract

- State transition in Tendermint is dependent on the messages received and timeouts.
- In other words, the state is changed in response to messages received by a processor or in the event of timeouts.
- The timeout mechanism ensures liveness and prevents endless waiting.
    - It is assumed that, eventually, after a period of asynchrony, there will be a round or communication period during which all processes can communicate in a timely fashion, which will ensure that processes eventually decide on a value.



Types of Messages in Tendermint

There are three types of messages in Tendermint:

1. **Proposal**: As the name suggests, this is used by the leader of the current round to propose a value or block.
2. **Pre-vote**: This message is used to vote on a proposed value.
3. **Pre-commit**: This message is also used to vote on a proposed value.

- These messages can be considered somewhat equivalent to PBFT's `PRE-PREPARE`, `PREPARE`, and `COMMIT` messages.
- Note that in Tendermint, only the proposal message carries the original value and the other two messages, pre-vote and pre-commit, operate on a value identifier, representing the original proposal.

Message Timeouts

All of the aforementioned messages also have a corresponding timeout mechanism, which ensures that processes do not end up waiting indefinitely for some conditions to meet. If a processor cannot decide in an expected amount of time, it will time out and trigger a round change.

Each type of message has an associated timeout. As such, there are three timeouts in Tendermint, corresponding to each message type:

1. `Timeout-propose`
2. `Timeout-prevote`
3. `Timeout-precommit`

These timeout mechanisms prevent the algorithm from waiting infinitely for a condition to be met. They also ensure that processes progress through the rounds.

- A clever mechanism to increase timeout with every new round ensures that after reaching GST, eventually the communication between correct processes becomes reliable and a decision can be reached.

State Variables in Tendermint

All processes in Tendermint maintain a set of variables, which helps with the execution of the algorithm. Each of these variables holds critical values, which ensure the correct execution of the algorithm.

- **Step**: The `step` variable holds information about the current state of the algorithm, that is, the current state of the Tendermint state machine in the current round.
- **lockedValue**: The `lockedValue` variable stores the most recent value (with respect to a round number) for which a pre-commit message has been sent.
- **lockedRound**: The `lockedRound` variable contains information about the last round in which the process sent a non-nil `pre-commit` message. This is the round where a possible decision value has been locked.
    - This means that if a proposal message and corresponding 2F + 1 messages have been received for a value in a round, then, due to the reason that 2F + 1 prevotes have already been received for this value, this is a possible decision value.

State Variables in Tendermint Cont.

- **validValue**: The role of the `validValue` variable is to store the most recent possible decision value.
- **validRound**: The `validRound` variable is the last round in which `validValue` was updated.

> `lockedvalue`, `lockedRound`, `validValue`, and `validRound` are reset to the initial values every time a decision is reached.

Apart from the preceding variables, a process also stores the current consensus instance (called *height* in Tendermint), and the current round number.

- These variables are attached to every message.
- A process also stores an array of decisions.
- Tendermint assumes a sequence of consensus instances, one for each height.

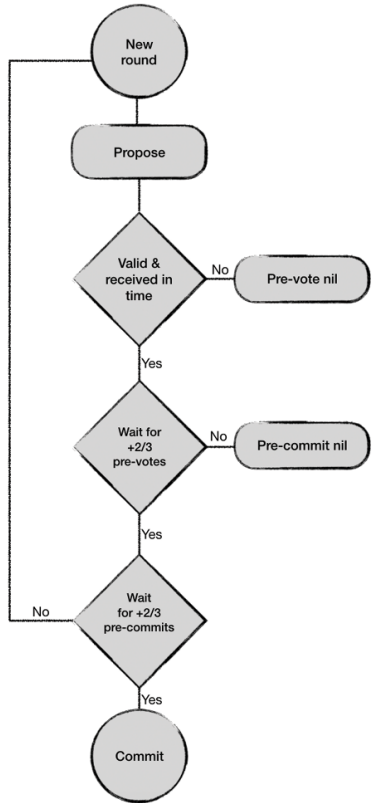
How Tendermint Works In More Detail

Tendermint works in rounds and each round comprises phases: **propose**, **pre-vote**, **pre-commit**, and **commit**.

1. Every round starts with a proposal value being proposed by a proposer. The proposer can propose any new value at the start of the first round for each height.

2. After the first round, any subsequent rounds will have the proposer, which proposes a new value only if there is no valid value present, that is, null.
    - Otherwise the `validValue`, that is, the possible decision value, is proposed, which has already been locked in a previous round.
    - The proposal message also includes a value of valid round, which denotes the last round in which there was a valid value updated.

3. The proposal is accepted by a correct process only if:
    1. The proposed value is valid
    2. The process has not locked on a round
    3. Or, the process has a value locked



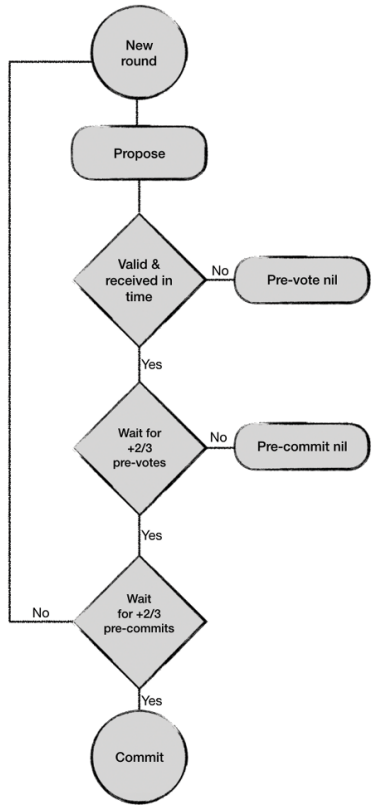
How Tendermint Works In More Detail Cont.

4. If the preceding conditions are met, then the correct process will accept the proposal and send a pre-vote message.

5. If the preceding conditions are not met, then the process will send a pre-vote message with a `` `nil` `` value.

6. In addition, there is also a timeout mechanism associated with the proposal phase, which initiates timeout if a process has not sent a pre-vote message in the current round or the timer expires in the proposal stage.

7. If a correct process receives a proposal message with a value and $2F + 1$ pre-vote messages, then it sends the pre-commit message.



How Tendermint Works In More Detail Cont.

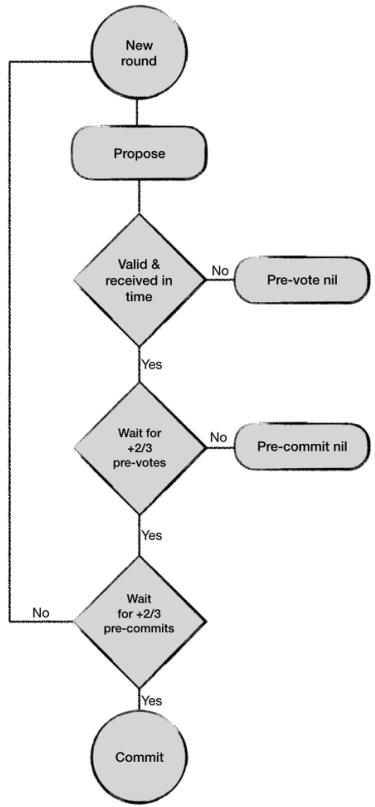
8. Otherwise, it sends out a `nil` pre-commit

9. A timeout mechanism associated with the pre-commit will initialize if the associated timer expires or if the process has not sent a pre-commit message after receiving a proposal message and 2F + 1 pre-commit messages.

10. A correct process decides on a value if it has received the proposal message in some round and 2F + 1 pre-commit messages for the ID of the proposed value.

11. There is also an associated timeout mechanism with this step, which ensures that the processor does not wait indefinitely to receive 2F + 1 messages. If the timer expires before the processor can decide, the processor starts the next round.

12. When a processor eventually decides, it triggers the next consensus instance for the next block proposal and the entire process of proposal, pre-vote, and pre-commit starts again.



Tendermint's Termination Mechanism

A new termination mechanism was introduced in Tendermint.

- For this purpose, there are two variables, namely `validValue` and `validRound`, which are used by the proposal message.
- Both of these variables are updated by a correct process when the process receives a valid proposal message and subsequent/corresponding $2F + 1$ pre-vote messages.

This process works by utilizing the gossip protocol, which ensures that if a correct process has locked a value in a round, all correct processes will then update their `validValue` and `validRound` variables with the locked values by the end of the round during which they have been locked.

- The key idea is that once these values have been locked by a correct processor, they will be propagated to other nodes within the same round and each processor will know the locked value and round, that is, the valid values.
- Now, when the next proposal is made, the locked values will be picked up by the proposer, which have already been locked as a result of the valid proposal and corresponding $2F + 1$ pre-vote messages.
    - This way, it can be ensured that the value that processes eventually decide upon is acceptable as specified by the validity conditions described above.

Nakamoto Consensus

Nakamoto consensus, or Proof of Work (PoW), was first introduced with Bitcoin in 2009.

> At a fundamental level, the PoW mechanism is designed to mitigate Sybil attacks, which facilitates consensus and the security of the network.

> A Sybil attack is a type of attack that aims to gain a majority influence on the network to control the network. Once a network is under the control of an adversary, any malicious activity could occur.

A Sybil attack is usually conducted by a node generating and using multiple identities on the network. If there are enough multiple identities held by an entity, then that entity can influence the network by skewing majority-based network decisions.

- *The majority in this case is held by the adversary.*

It is quite easy to obtain multiple identities and try to influence the network.

- However, in Bitcoin, due to the hashing power requirements, this attack is mitigated.

How PoW Works

In summary, PoW works like this:

- PoW makes use of hash puzzles.
- A node proposes a block has to find a nonce such that *H(nonce || previous hash || Tx || Tx|| . . . ||Tx)* < *Threshold value*.

PoW Key Philosophy

The key idea behind PoW as a solution to the Byzantine generals problem is that all honest generals (miners in the Bitcoin world) achieve agreement on the same state (decision value).

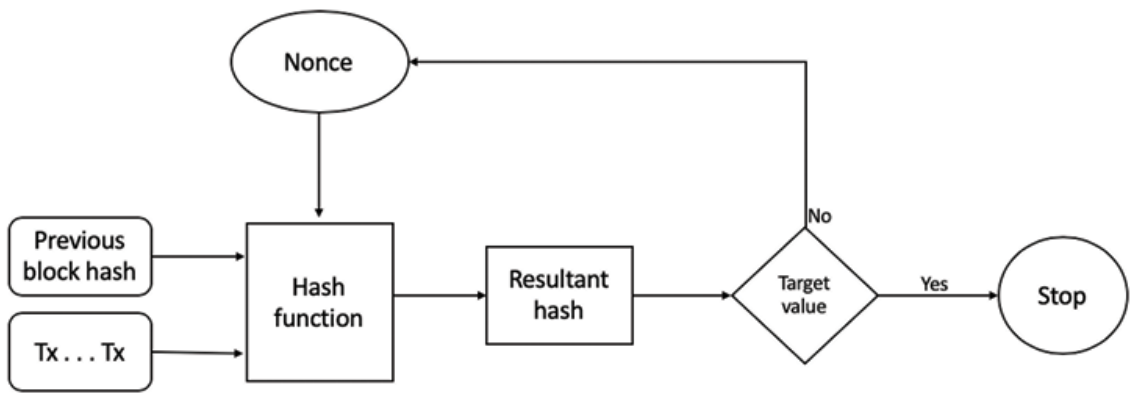
- As long as honest participants control the majority of the network, PoW solves the Byzantine generals problem.
    - Note that this is a probabilistic solution and not deterministic.

> Original post by Satoshi Nakamoto can be found here.

How PoW Works Cont.

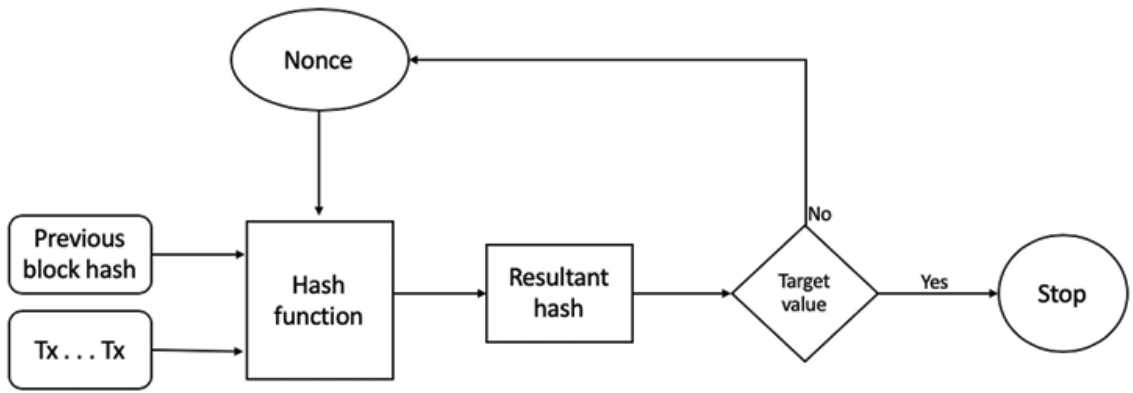
The process can be summarized as follows:

1. New transactions are broadcast to all nodes on the network.
2. Each node collects the transactions into a candidate block.
3. Miners propose new blocks.
4. Miners concatenate and hash with the header of the previous block.



How PoW Works Cont.

5. The resultant hash is checked against the target value, that is, the network difficulty target value.
6. If the resultant hash is less than the threshold value, then PoW is solved, otherwise, the nonce is incremented and the node tries again. This process continues until a resultant hash is found that is less than the threshold value.



Nakamoto vs. Traditional Consensus

Why should we even compare the two, you might ask?

Nakamoto consensus was the first of its kind.

It solved the consensus problem, which had been traditionally solved using pre-Bitcoin protocols like PBFT.

A natural question that arises is, are there any similarities between the Nakamoto world and traditional distributed consensus?

- The short answer is yes, because we can map the properties of PoW consensus to traditional Byzantine consensus.
    - This mapping is useful to understand what properties of traditional consensus can be applied to the Nakamoto world and vice versa.

In traditional consensus algorithms, we have **agreement**, **validity**, and **liveness** properties, which can be mapped to Nakamoto-specific properties of the **common prefix**, **chain quality**, and **chain growth** properties respectively.

Nakamoto Consensus Properties

- The **common prefix** property means that the blockchain hosted by honest nodes will share the same large common prefix.
    - If that is not the case, then the **agreement** property of the protocol cannot be guaranteed, meaning that the processors will not be able to decide and agree on the same value.
- The **chain quality** property means that the blockchain contains a certain required level of correct blocks created by honest nodes (miners).
    - If chain quality is compromised, then the **validity** property of the protocol cannot be guaranteed. This means that there is a possibility that a value will be decided that is not proposed by a correct process, resulting in safety violation.
- The **chain growth** property simply means that new correct blocks are continuously added to the blockchain.
    - If chain growth is impacted, then the **liveness** property of the protocol cannot be guaranteed. This means that the system can deadlock or fail to decide on a value.

Variants of PoW

- **CPU-bound PoW**: CPU-bound PoW refers to a type of PoW where the processing required to find the solution to the cryptographic hash puzzle is directly proportional to the calculation speed of the CPU or hardware such as ASICs.
    - Because ASICs have dominated the BitcoinPoW and provide somewhat undue advantage to the miners who can afford to use ASICs, this CPU-bound PoW is seen as shifting toward centralization.
    - Moreover, mining pools with extraordinary hashing power can shift the balance of power towards them.
    - Therefore, memory-bound PoW algorithms have been introduced, which are ASIC-resistant and are based on memory-oriented design instead of CPU.

- **Memory-bound PoW**: Memory-bound PoW algorithms rely on system RAM to provide PoW.
    - Here, the performance is bound by the access speed of the memory or the size of the memory.
    - This reliance on memory also makes these PoW algorithms ASIC-resistant.
    - Equihash is one of the most prominent memory-bound PoW algorithms.

Proof of Stake (PoS)

> PoS is an energy-efficient alternative to the PoW algorithm, which consumes an enormous amount of energy.

- The stake represents the number of coins (money) in the consensus protocol staked by a blockchain participant.
    - The key idea is that if someone has a stake in the system, then they will not try to sabotage the system.
    - Generally speaking, the chance of proposing the next block is directly proportional to the value staked by the participant.

There are different variations of PoS, such as **chain-based PoS**, **committee-based PoS**, **BFT-based PoS**, **delegated PoS**, **leased PoS**, and **master node PoS**.

How PoS Works

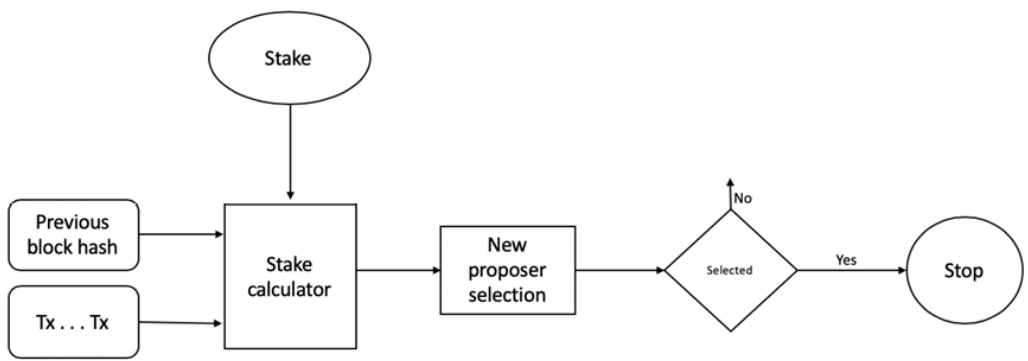
In PoS systems, there is no concept of mining in the traditional Nakamoto consensus sense.

However, the process related to earning revenue is sometimes called **virtual mining**.

- A PoS miner is called either a **validator**, **minter**, or **stakeholder**.

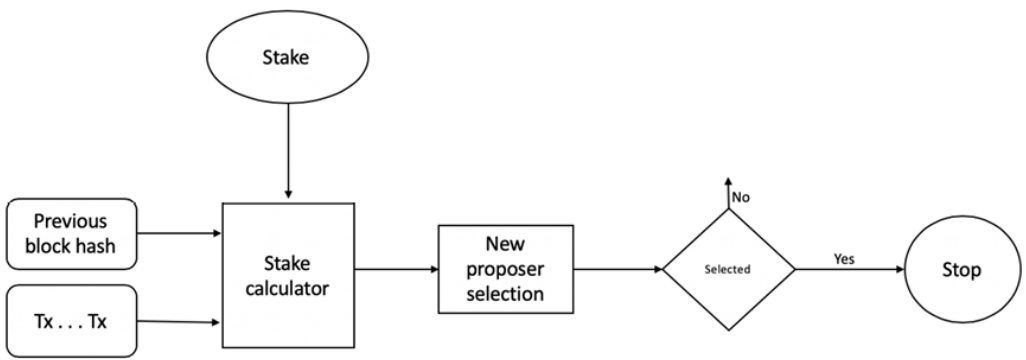
The right to win the next proposer role is usually assigned randomly.

- Proposers are rewarded either with transaction fees or block rewards.
- Similar to PoW, control over the majority of the network in the form of the control of a large portion of the stake is required to attack and control the network.



How PoS Works Cont.

- PoS mechanisms generally select a stakeholder and grant appropriate rights to it based on their staked assets.
- The stake calculation is application-specific, but generally, is based on balance, deposit value, or voting among the validators.
- Once the stake is calculated, and a stakeholder is selected to propose a block, the block proposed by the proposer is readily accepted.
- The probability of selection increases with a higher stake.
    - In other words, the higher the stake, the better the chances of winning the right to propose the next block.



Types of PoS: Chain-based PoS

This mechanism is very similar to PoW. The only change from the PoW mechanism is the block generation method.

A block is generated in two steps by following a simple protocol:

- Transactions are picked up from the memory pool and a candidate block is created.
- A clock is set up with a constant tick interval, and at each clock tick, whether the hash of the block header concatenated with the clock time is less than the product of the target value and stake value is checked. This process can be shown in a simple formula:

$$Hash(B||clocktime) < targetxstakevalue$$

Types of PoS: Chain-based PoS Cont.

The stake value is dependent on the way the algorithm is designed.

In some systems, it is directly proportional to the amount of stake, and in others, it is based on the amount of time the stake has been held by the participant (also called *coinage*).

- The target is the mining difficulty per unit of the value of stake.

This mechanism still uses hashing puzzles, as in PoW.

- But, instead of competing to solve the hashing puzzle by consuming a high amount of electricity and specialized hardware, the hashing puzzle in PoS is solved at regular intervals based on the clock tick.
- A hashing puzzle becomes easier to solve if the stake value of the minter is high.

Types of PoS: Committee-based PoS

- In this scheme, a group of stakeholders is chosen randomly, usually by using a **verifiable random function (VRF)**.
- This VRF, once invoked, produces a random set of stakeholders based on their stake and the current state of the blockchain.
- The chosen group of stakeholders becomes responsible for proposing blocks in sequential order.
- This mechanism is used in the Ouroboros PoS consensus mechanism, which is used in Cardano

Types of PoS: Delegated PoS

DPoS is very similar to committee-based PoS, but with one crucial difference.

- Instead of using a random function to derive the group of stakeholders, the group is chosen by stake delegation.
- The group selected is a fixed number of minters that create blocks in a round- robin fashion.
- Delegates are chosen via voting by network users. Votes are proportional to the amount of the stake that participants have on the network.
    - This technique is used in Lisk, Cosmos, and EOS.
- DPoS is not decentralized as a small number of known users are made responsible for proposing and generating blocks.

HotStuff

HotStuff is a part of the latest class of BFT protocols with a number of optimizations. There are several changes in HotStuff that make it a different and, in some ways, better protocol than traditional PBFT.

There are three key properties that HotStuff has addressed. These properties are listed as follows:

1. **Linear view change**

Linear view change results in reduced communication complexity.

- It is achieved by the algorithm where after GST is reached, a correct designated leader will send only $O(n)$ **authenticators** (either a partial signature or signature) to reach consensus.
- In the worst case, where leaders fail successively, the communication cost is $O(n^2)$ - quadratic.
    - In simpler words, quadratic complexity means that the performance of the algorithm is proportional to the squared size of the input.

HotStuff Cont.

2. **Optimistic responsiveness**

- Optimistic responsiveness ensures that any correct leader after GST is reached only requires the first $N - F$ responses to ensure progress.

3. **Chain quality**

- This property ensures fairness and liveness in the system by allowing fast and frequent leader rotation.

- *All these properties together have been addressed for the first time in the HotStuff protocol.*

HotStuff Innovations Cont.

Another innovation in HotStuff is the separation of safety and liveness mechanisms.

- The separation of concerns allows better modularity, cleaner architecture, and control over the development of these features independently.

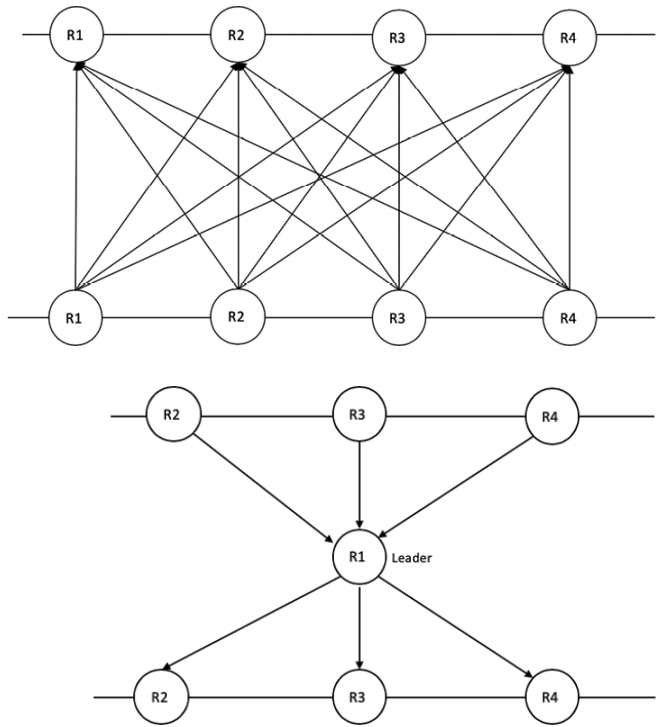
Safety is ensured through voting and commit rules for participant nodes in the network.

Liveness, on the other hand, is the responsibility of a separate module, called **Pacemaker**, which ensures a new, correct, and unique leader is elected.

HotStuff vs. PBFT

In comparison with traditional PBFT, HotStuff has introduced several changes, which result in improved performance:

1. PBFT-style protocols work using a mesh communication topology, where each message is required to be broadcast to other nodes on the network.
    - In HotStuff, the communication has been changed to the star topology, which means that nodes do not communicate with each other directly, but all consensus messages are collected by a leader and then broadcast to other nodes.
    - This immediately results in reduced communication complexity.



HotStuff vs. PBFT Cont.

However, a question arises here: what happens if the leader somehow is corrupt or compromised?

- This issue is solved by the same BFT tolerance rules where, if a leader proposes a malicious block, it will be rejected by other honest validators and a new leader will be chosen.
- This scenario can slow down the network for a limited time (until a new honest leader is chosen), but eventually (as long as a majority of the network is honest), an honest leader will be chosen, which will propose a valid block.
- Also, for further protection, usually, the leader role is frequently (usually, with each block) rotated between validators, which can neutralize any malicious attacks targeting the network.
- This property ensures fairness, which helps to achieve chain quality, introduced previously.

HotStuff vs. PBFT Cont.

2. PBFT has two main sub-protocols, namely **normal mode** and **view-change mode**.

- View-change mode is triggered when a leader is suspected of being faulty.
    - This approach does work to provide a liveness guarantee to PBFT but increases communication complexity significantly.
- HotStuff addresses this by merging the view-change process with normal mode.
    - This means that nodes can switch to a new view directly without waiting for a threshold of view-change messages to be received by other nodes.
- In PBFT, nodes wait for $2F + 1$ messages before the view change can occur, but in HotStuff, view change can occur directly without requiring a new sub-protocol.
    - Instead, the checking of the threshold of the messages to change the view becomes part of the normal view.

How HotStuff Works

The system is based on a standard BFT assumption of $N = 3F + 1$ nodes in the system, where F is a faulty node and N is the number of nodes in the network.

- Nodes communicate with each other via point-to- point message-passing, utilizing reliable and authenticated communication links.
- The network is supposed to be partially synchronous.

HotStuff makes use of threshold signatures where a single public key is used by all nodes, but a unique private key is used by each replica.

- The use of threshold signatures results in the decreased communication complexity of the protocol.
- In addition, cryptographic hash functions are used to provide unique identifiers for messages.

HotStuff Phases

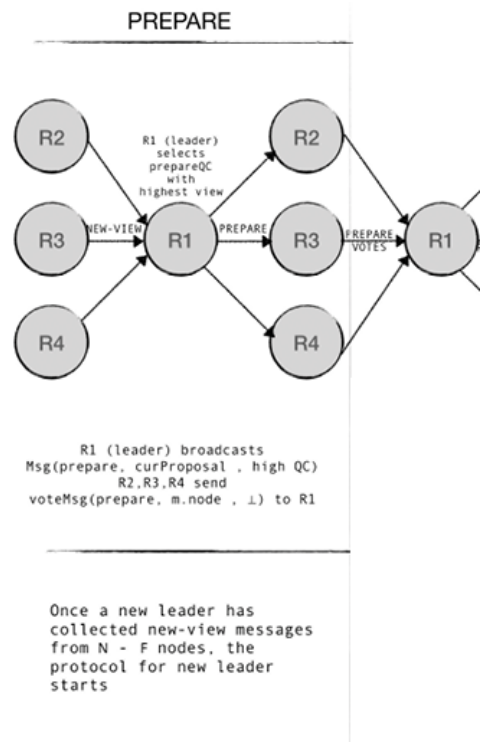
> HotStuff works in phases, namely the **prepare** phase, **pre-commit** phase, **commit** phase, and **decide** phase.

> A common term that we will see in the following section is quorum certificate. It is a data structure that represents a collection of signatures produced by $N-F$ replicas to demonstrate that the required threshold of messages has been achieved. In other words, it is simply a set of votes from $N-F$ nodes.

1. **Prepare**

Once a new leader has collected **new-view** messages from $N - F$ nodes, the protocol for the new leader starts.

- The leader collects and processes these messages to figure out the latest branch in which the highest quorum certificate of prepare messages was formed.



HotStuff Phases Cont.

2. **Pre-commit**

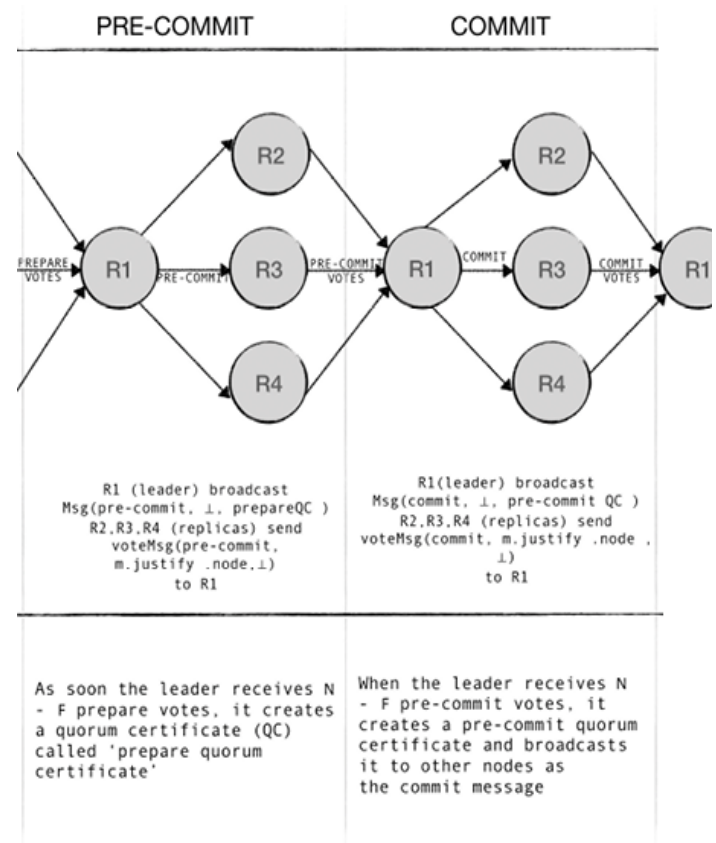
As soon as a leader receives $N - F$ prepare votes, it creates a quorum certificate called *"prepare quorum certificate."*

- This "prepare quorum certificate" is broadcast to other nodes as a `PRE-COMMIT` message.
- When a replica receives the `PRE-COMMIT` message, it responds with a pre-commit vote.
  - The quorum certificate is the indication that the required threshold of nodes has confirmed the request.

3. **Commit**

When the leader receives $N - F$ pre-commit votes, it creates a `PRE-COMMIT` quorum certificate and broadcasts it to other nodes as the `COMMIT` message.

- When replicas receive this `COMMIT` message, they respond with their commit vote.
- At this stage, replicas lock the `PRE-COMMIT` quorum certificate to ensure the safety of the algorithm even if view change-occurs.

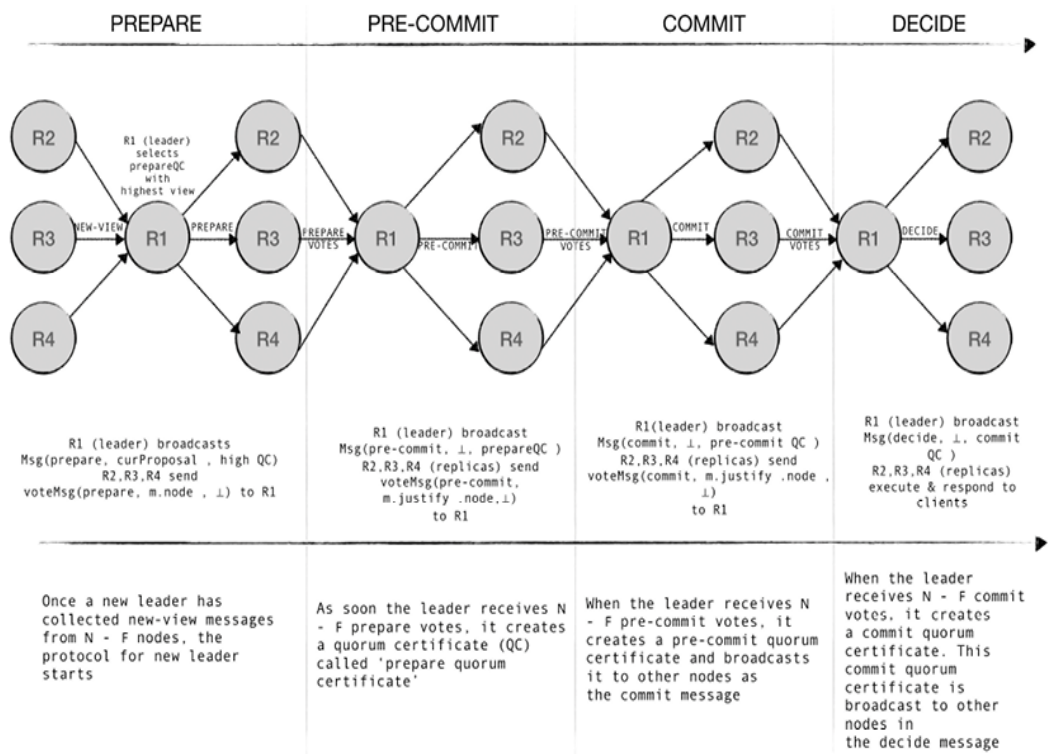


HotStuff Phases Cont.

4. **Decide**

When the leader receives N - F commit votes, it creates a `COMMIT` quorum certificate.

- This `COMMIT` quorum certificate is broadcast to other nodes in the `DECIDE` message.
- When replicas receive this `DECIDE` message, replicas execute the request, because this message contains an already committed certificate/value.
- Once the state transition occurs as a result of the `DECIDE` message being processed by a replica, the new view starts.



HotStuff's Liveness: Pacemaker

> HotStuff guarantees liveness (progress) by using **Pacemaker**, which ensures progress after GST within a bounded time interval.

This component has two elements:

- There are time intervals during which all replicas stay at a height for sufficiently long periods.
  - This property can be achieved by progressively increasing the time until progress is made (a decision is made).
- A unique and correct leader is elected for the height.
  - New leaders can be elected deterministically by using a rotating leader scheme or pseudo-random functions.

Safety in HotStuff is guaranteed by voting and relevant commit rules.

Liveness and safety are separate mechanisms that allow independent development, modularity, and separation of concerns.

HotStuff Summary

- HotStuff is a simple yet powerful protocol that provides linearity and responsiveness properties.
- It allows consensus without any additional latency, at the actual speed of the network.
- Moreover, it is a protocol that manifests linear communication complexity, thus reducing the cost of communication compared to PBFT-style protocols.
- It is also a framework in which other protocols, such as DLS, PBFT, and Tendermint can be expressed.

So, which algorithm should we choose?

Choosing a consensus algorithm depends on several factors.

It is not only use case-dependent, but some trade-offs may also have to be made to create a system that meets all the requirements without compromising the core safety and liveness properties of the system.

Additionally, there are case-specific factors such as **finality**, **speed**, **performance** and **scalability** that we might need to take into account.

Finality

> Finality refers to a concept where once a transaction has been completed, it cannot be reverted.

- In other words, if a transaction has been committed to the blockchain, it won't be revoked or rolled back.
- This feature is especially important in financial networks, where once a transaction has gone through, the consumer can be confident that the transaction is irrevocable and final.
- There are two types of finality, **probabilistic** and **deterministic**.

Probabilistic Finality

> Probabilistic finality, as the name suggests, provides a probabilistic guarantee of finality. The assurance that a transaction, once committed to the blockchain, cannot be rolled back builds over time instead of immediately.

For example, in Nakamoto consensus, the probability that a transaction will not be rolled back increases as the number of blocks after the transaction commits increases.

As the chain grows, the block containing the transaction goes deeper, which increasingly ensures that the transaction won't be rolled back.

While this method worked and stood the test of time for many years, it is quite slow.

- For example, in the Bitcoin network, users usually have to wait for six blocks, which is equivalent to an hour, to get the right level of confidence that a transaction is final. This type of finality might be acceptable in public blockchains.
- Still, such a delay is not acceptable in financial transactions in a consortium blockchain. In consortium networks, we need immediate finality.

Deterministic Finality

> Deterministic finality or immediate finality provides an absolute finality guarantee for a transaction as soon as it is committed in a block.

- There are no forks or rollbacks, which could result in a transaction rollback.
- The confidence level of transaction finality is 100 percent as soon as the block that contains the transaction is finalized.
- This type of finality is provided by fault-tolerant algorithms such as PBFT.

Speed, performance, and scalability

Performance is a significant factor that impacts the choice of consensus algorithms.

- PoW chains are slower than BFT-based chains.
    - If performance is a crucial requirement, then it is advisable to use voting-based algorithms for permissioned blockchains such as PBFT, which will provide better performance in terms of quicker transaction processing.
    - There is, however, a caveat that needs to be kept in mind here—PBFT-type blockchains do not scale well but provide better performance.
    - On the other hand, PoW-type chains can scale well but are quite slow and do not meet the enterprise-grade performance requirements.