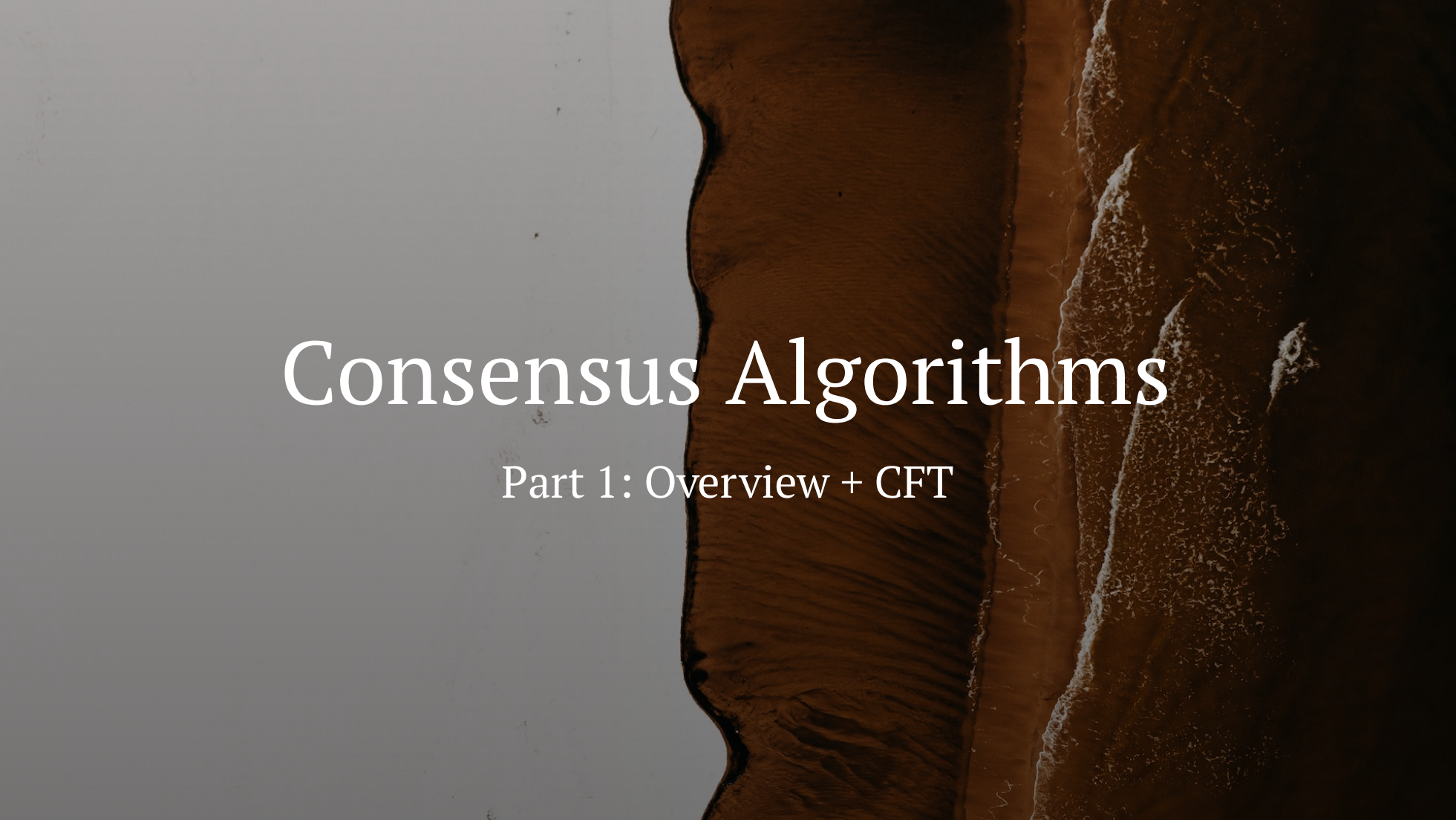


Consensus Algorithms

Part 1: Overview + CFT

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 a ack 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.

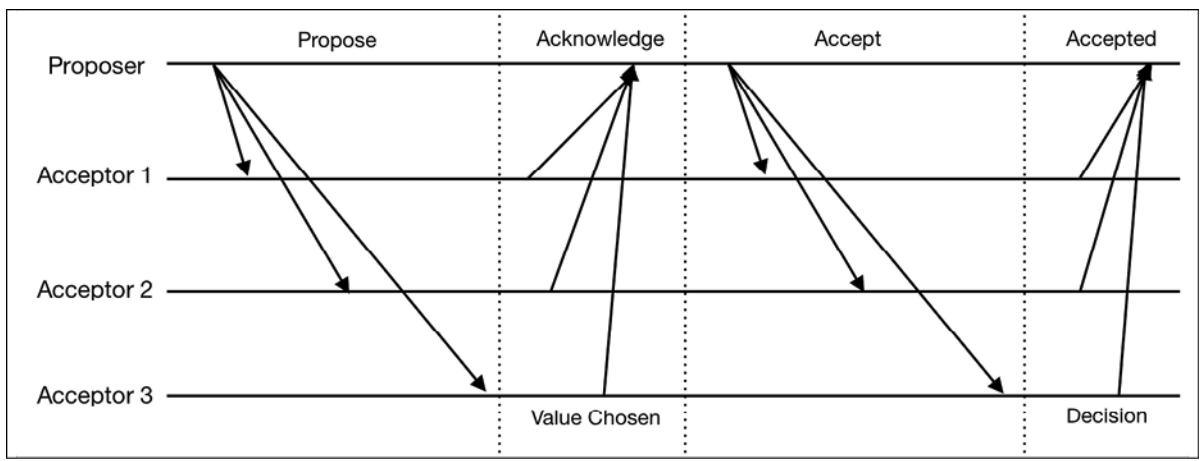
4. If the majority of the acceptors accept the proposed value (now the "accept" message), then it is decided: that is, agreement is achieved.

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)

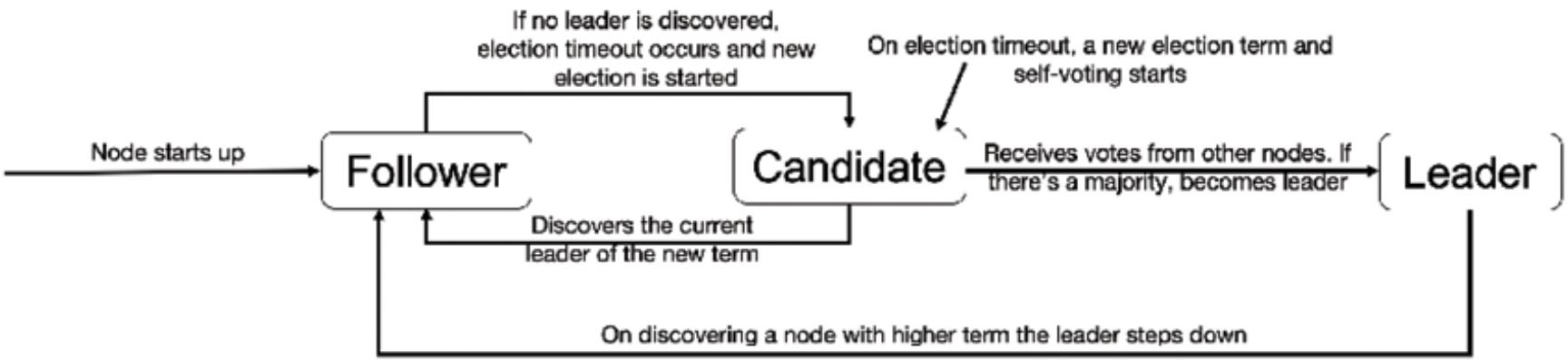
- Each server in Raft can have either a **follower**, **leader**, or **candidate** state.
- The protocol ensures election **safety** (that is, only one winner each election term) and **liveness** (that is, some candidate must eventually win).

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.

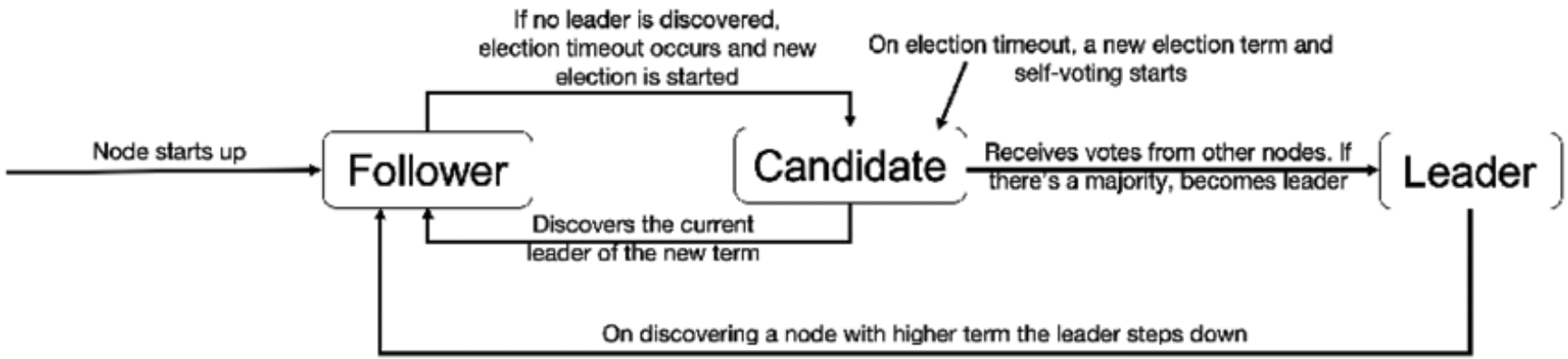


How Raft Works Cont.

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.

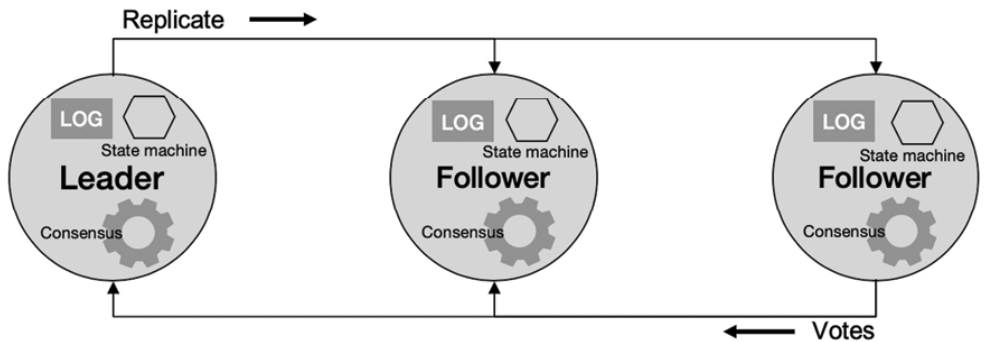
5. The leader notifies the followers regarding the committed entry.

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.

Onto the next part: BFT Algorithms!