5
Consensus Algorithms

Consensus is a fundamental problem in distributed systems. Since the 1970s, this problem has been researched in the context of distributed systems, but a renewed interest has arisen in developing distributed consensus algorithms that are suitable for blockchain networks. In this chapter, we will explore the underlying techniques behind distributed consensus algorithms, their inner workings, and new algorithms that have been specifically developed for blockchain networks.

In addition, we will introduce various well-known algorithms in a traditional distributed systems arena that can also be implemented in blockchain networks with some modifications, such as Paxos, Raft, and PBFT. We will also explore other mechanisms that have been introduced specifically for blockchain networks such as **proof of work** (**PoW**), **proof of stake** (**PoS**), and modified versions of traditional consensus such as **Istanbul Byzantine Fault Tolerant** (**IBFT**), which is a modified, '**blockchained**' version of the **Practical Byzantine Fault Tolerant** (**PBFT**) algorithm. Along the way, we'll cover the following topics:

- Introducing consensus
- Analysis and design
- Classification
- Algorithms
- Choosing an algorithm

Before we delve into specific algorithms, we first need to understand some fundamental concepts and an overview of the consensus problem.

Introducing consensus

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

classified into two main categories, namely, **message passing** and **shared memory**. 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. Consensus is the process that allows all processes in a network to agree on some specific value in the presence of faults.

As we saw in *Chapter 1*, *Blockchain 101*, there are different types of blockchain networks. In particular, two types, permissioned and public (permissionless), were discussed. The consensus problem can also be classified based on these two paradigms. For example, Bitcoin is a public blockchain. It runs PoW, also called **Nakamoto consensus**. In contrast, many permissioned blockchains tend to run variants of traditional or classical distributed consensus. A prime example is IBFT, which is a **blockchained** version of PBFT. Other examples include Tendermint, Casper FFG, and many variants of PBFT.

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

Fault tolerance

A fundamental requirement in a consensus mechanism is fault tolerance in a network, and it should continue to work even in the presence of faults. This naturally means that there must be some limit to the number of faults a network can handle since no network can operate correctly if many of its nodes fail. Based on the fault-tolerance requirement, consensus algorithms are also called fault-tolerant algorithms, and there are two types of fault-tolerant algorithms:

- **Crash fault-tolerance** (**CFT**), which covers only crash faults or, in other words, benign faults.
- **Byzantine fault-tolerance** (**BFT**), which deals with the type of faults that are arbitrary and can even be malicious.

Replication is a standard approach to improve the fault tolerance and availability of a network. Replication results in a synchronized copy of

data across all nodes in a 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:

- **Active replication**, which is a type where each replica becomes a copy of the original state machine replica.
- **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.

In the context of fault-tolerant consensus mechanisms, replication plays a vital role by introducing resiliency into the system.

State machine replication (SMR) is a de facto technique that is used to provide deterministic replication services to achieve fault tolerance in a distributed system. At an abstract level, a state machine is a mathematical model that is used to describe a machine that can be in different states but occupies 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 the input received. As a result of state transition, an output is produced along with an updated state. The fundamental 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 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.

Now that we understand the basics of replication and fault tolerance, it is important to understand 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.

FLP impossibility

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.

The problems that are not solvable under any conditions are known as **unsolvability results**. This result is known as the **FLP impossibility** result, which is a fundamental unsolvability result that states that in an asynchronous environment, the deterministic consensus is impossible, even if only one process is faulty.

> FLP is named after the authors' names, Fischer, Lynch, and Patterson. Their result was presented in their paper:

Fischer, M.J., Lynch, N.A., and Paterson, M.S., 1982. *Impossibility of distributed consensus with one faulty process (No. MIT/LCS/TR-282). Massachusetts Inst of Tech Cambridge lab for Computer Science.*

The paper is available at https://apps.dtic.mil/dtic/tr/fulltext/u2/a132503.pdf .

To circumvent **FLP impossibility**, several techniques have been introduced:

- **Failure detectors** can be seen as **oracles** associated with processors to detect failures. In practice, usually, this is implemented as a timeout mechanism.
- **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. Eventually, with a very high probability, the entire network flips to a decision value.
- **Synchrony assumptions**, where additional synchrony and timing assumptions are made to ensure that the consensus algorithm gets adequate time to run so that it can make progress and terminate.

Now that we understand a fundamental impossibility result, let's look at another relevant result that highlights the unsolvability of consensus due to a lack of resources: that is, a **lower bound result**. We can think of lower bound as a minimum number of resources, for example, the number of processors or communication links required to solve a problem. The most common and fundamental of these results is the minimum number of processors required for consensus. These results are listed below, where *f* represents the number of failed nodes:

- 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.

We have now covered the fundamentals of distributed consensus theory. Now, we'll delve a little bit deeper into the analysis and design of consensus algorithms.

Analysis and design

To analyze and understand a consensus algorithm, we need to define a model under which our algorithm will run. This model provides some assumptions about the operating environment of the algorithm and provides a way to intuitively study and reason about the various properties of the algorithm.

In the following sections, we'll describe a model that is useful for describing and analyzing consensus mechanisms.

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. We will now look at this model in detail and introduce all objects one by one.

Processes

Processes communicate with each other by passing messages to each other. Therefore, these systems are called message-passing distributed systems. There is another class, called shared memory, which we will not discuss here, as all blockchain systems are message-passing systems.

Timing assumptions

There are also some timing assumptions that are made when designing consensus algorithms:

- **Synchrony**: In synchronous systems, there is a known upper bound on 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 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.
- **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. An eventually synchronous system is a type of partial synchrony, which means that the system becomes synchronous after an instance of time called **global stabilization time** (**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.

Now that we understand the fundamentals of the distributed consensus theory, let's look at two main classes of consensus algorithms. This categorization emerged after the invention of Bitcoin. Prior to Bitcoin, there was a long history of research in distributed consensus protocols.

Classification

The consensus algorithms can be classified into two broad categories:

- **Traditional**: Voting-based consensus. Traditional voting-based consensus has been researched in distributed systems for many decades. Many fundamental results and a lot of ground-breaking work have al-

ready been produced in this space. Algorithms like **Paxos** and **PBFT** are prime examples of such types of algorithms. Traditional consensus can also be called fault-tolerant distributed consensus.
- **Lottery-based**: Nakamoto and post-Nakamoto consensus. Lottery-based or Nakamoto-type consensus was first introduced with Bitcoin. This class can also be simply called blockchain consensus.

> Note that they are distinguished by the time period in which they've been invented; traditional protocols were developed before the introduction of Bitcoin, and lottery-based protocols were introduced with Bitcoin.

The fundamental requirements of consensus algorithms boil down to **Safety** and **Liveness** conditions. The **Safety** requirement generally means that nothing bad happens, and is usually based on some safety requirements of the algorithms, such as agreement, validity, and integrity. 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 on 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.

The **Liveness** requirement generally means that something good eventually happens. It means that the protocol can make progress even if the network conditions are not ideal. It usually has one requirement:

- **Termination**. This liveness property states that each honest node must eventually decide on a value.

With this, we have covered the classification and requirements of consensus algorithms. In the next section, we'll introduce various consensus al-

gorithms that we can evaluate using the consensus models covered in this section.

Algorithms

In this section, we will discuss the key algorithms in detail. We'll be looking at the two main types of fault-tolerant algorithms, which are classified based on the fault tolerance level they provide: CFT or BFT.

CFT algorithms

We'll begin by looking at some algorithms that solve the consensus problem with crash fault tolerance. One of the most fundamental algorithms in this space is Paxos.

Paxos

Leslie Lamport developed Paxos. It is the most fundamental distributed consensus algorithm, allowing 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. There are many other protocols that have since emerged from basic Paxos, such as Multi-Paxos, Fast Paxos, and Cheap Paxos.

> Paxos was proposed first in 1989 and then later, more formally, in 1998, in the following paper:
>
> Lamport, L., 1998. "The part-time parliament", *ACM Transactions on Computer Systems (TOCS)*, 16(2): pp. 133-169.
>
> The paper is available here: `https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf`.

Paxos works under an asynchronous network model and supports the handling of only benign failures. This is not a Byzantine fault-tolerant protocol. However, later, a variant of Paxos was developed that provides BFT.

The paper in which a Byzantine fault-tolerant version of Paxos is described is available here:

Lamport, L., 2011, September. *Byzantizing Paxos by refinement*. *International Symposium on Distributed Computing* (pp. 211-224). Springer, Berlin, Heidelberg.

A link to the paper is available here: `http://lamport.azurewebsites.net/pubs/web-byzpaxos.pdf`.

Paxos makes use of *2f + 1* processes to ensure fault tolerance in a network where processes can crash fault, that is, experience benign failures. Benign failure means either the loss of a message or a process stops. In other words, Paxos can tolerate one crash failure in a three-node network.

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 a proposer and acceptors as participants, where the proposer is the replicas or nodes that propose the values, and acceptors are the nodes that accept the values.

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.
- **Validity**, which means that only the proposed values are decided.

Under liveness, we have:

- **Termination**, which means that, eventually, the protocol is able to decide and terminate.

Processes can assume different roles, which are listed as follows:

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

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.

Paxos can be seen as a protocol that is quite similar to 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. 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 improvements contribute toward ensuring the safety and liveness of the Paxos algorithm.

We'll now describe how the Paxos protocol 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.

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

We can visualize this process in the following diagram:

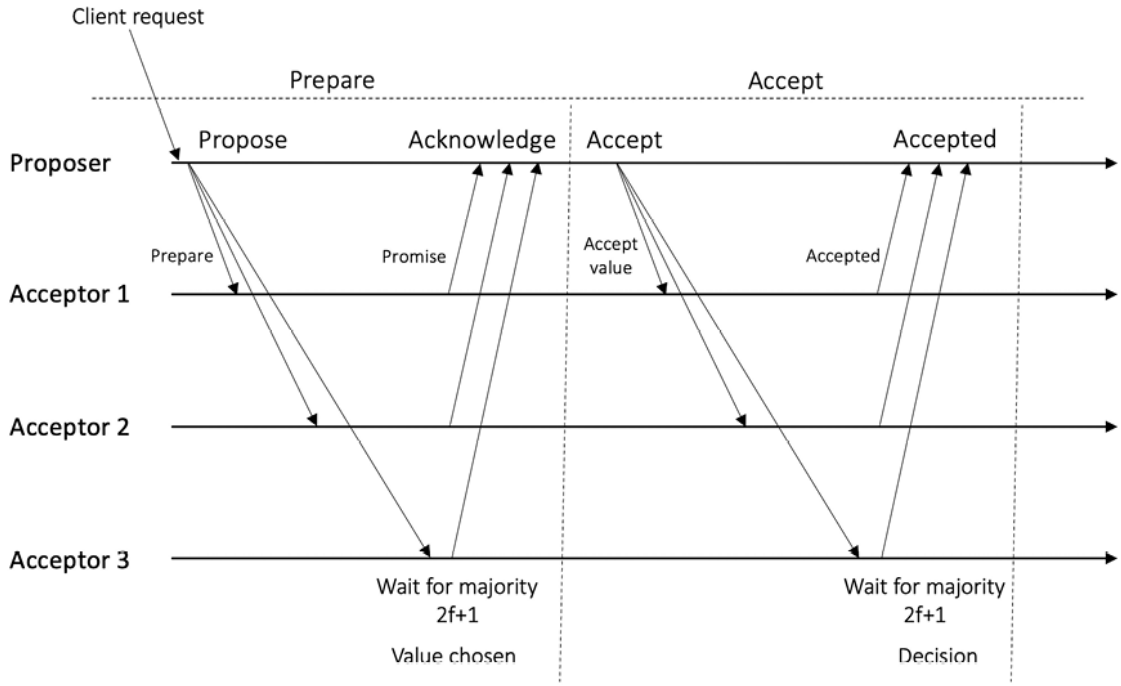

Figure 5.1: How Paxos works

A natural question arises about how Paxos ensures its safety and liveness guarantees. Paxos, at its core, is quite simple, yet it achieves all these properties efficiently. The actual proofs for the correctness of Paxos are quite in-depth and are not the subject of this chapter. However, the rationale behind each property is presented 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 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.

Even though the Paxos algorithm is quite simple at its core, it is seen as challenging to understand, and many academic papers have been written to explain it. This slight problem, however, has not prevented it from being implemented in many production networks, such as Google's Spanner, as it has proven to be the most efficient protocol to solve the consensus problem. Nevertheless, there have been attempts to create alternative easy-to-understand algorithms. Raft is such an attempt to create an easy-to-understand CFT algorithm.

Raft

The Raft protocol is a CFT consensus mechanism developed by Diego Ongaro and John Ousterhout at Stanford University. In Raft, 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 subproblems:

- **Leader election** (a new leader election in case the existing one fails)
- **Log replication** (leader-to-follower log sync)
- **Safety** (no conflicting log entries (index) between servers)

The Raft protocol ensures **election safety** (that is, only one winner each election term), **leader append-only**, **log matching**, **leader completeness**, **liveness** (that is, some candidate must eventually win), and **state machine safety**.

Each server in Raft can have either a **follower**, **leader**, or **candidate** state. At a fundamental level, the protocol is quite simple and can be described simply by the following sequence:

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.
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.

The state transition of the Raft algorithm can be visualized in the following diagram:

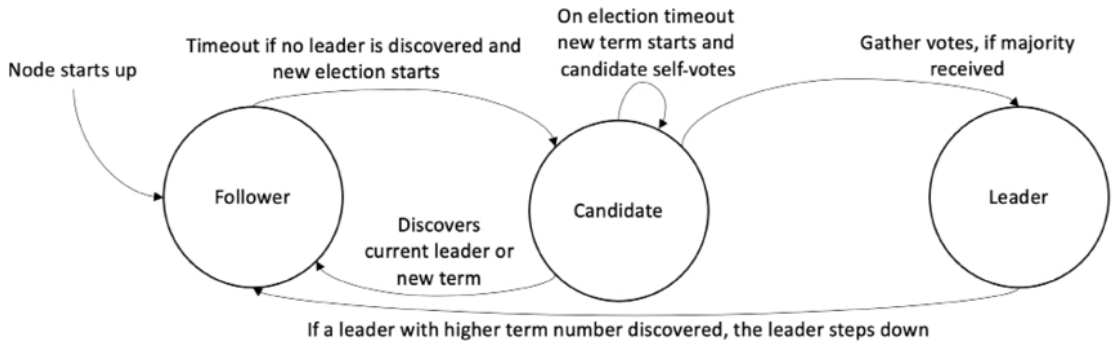

Figure 5.2: Raft state transition

We saw earlier that data is eventually replicated across all nodes in a consensus mechanism. In Raft, the log (data) is eventually replicated across all nodes. The replication logic can be visualized in the following diagram. The aim of log replication is to synchronize nodes with each other:

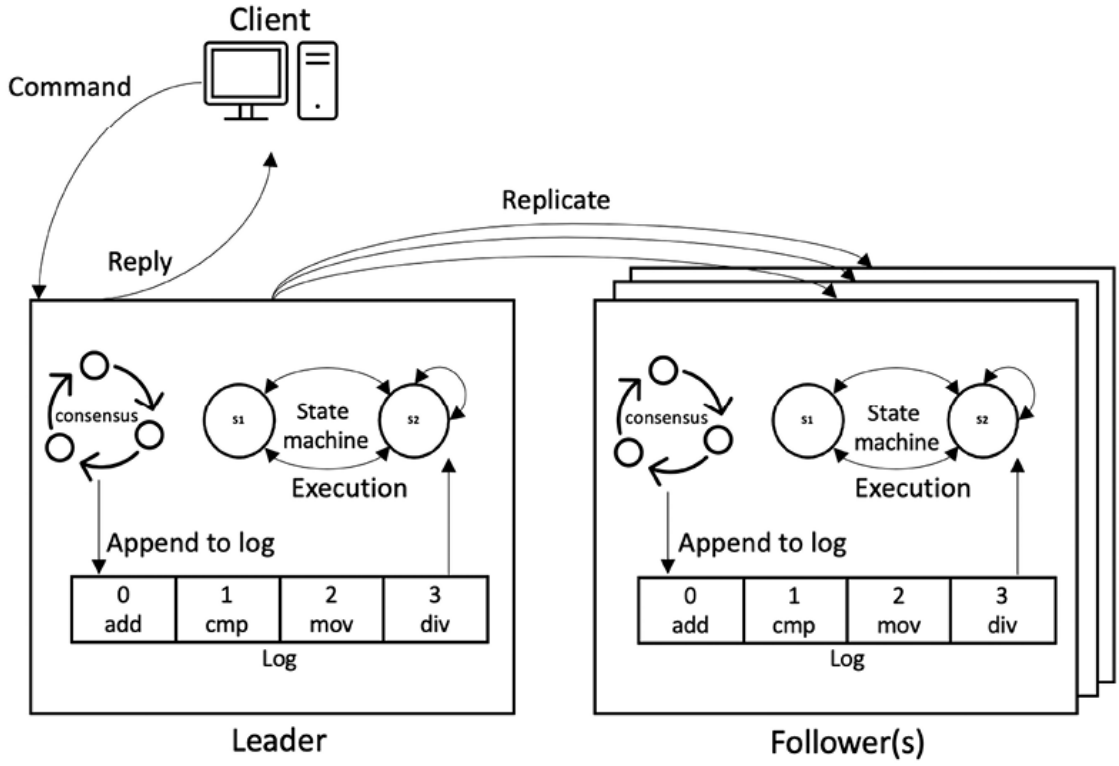

Figure 5.3: Log replication mechanism

Log replication is a simple mechanism. 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.

With this, our discussion on CFT algorithms is complete. Now we'll introduce Byzantine fault-tolerant algorithms, which have been a research area for many years in distributed computing.

BFT algorithms

In this section, we'll introduce the mechanisms that were developed to solve the Byzantine generals (consensus in the presence of faults) problem.

Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (**PBFT**) was developed in 1999 by Miguel Castro and Barbara Liskov. PBFT, as the name suggests, is a protocol developed to provide consensus in the presence of Byzantine faults. This algorithm demonstrated for the first time that PBFT is possible.

PBFT comprises three subprotocols:

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

The protocol runs in rounds where, in each round, an elected leader, or primary, node handles the communication with the client. In each round, the protocol progresses through three phases or steps. These phases are pre-prepare, prepare, and commit.

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 act as backups. Each replica maintains a local state comprising three main elements: a service state, a message log, and a number representing the replica's current view.

PBFT is based on the SMR protocol introduced earlier. 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 BFT as long as the number of nodes in a system stays $n >= 3f + 1$.

Now we'll discuss the phases mentioned above one by one, starting with pre-prepare.

Pre-prepare: The main purpose of this phase is to assign a unique sequence number to the request. We can think of it as an orderer. This is the first phase in the protocol, where the primary node, or **primary**, receives (accepts) 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:

1. First, whether the digital signature is valid.
2. After this, whether the current view number is valid.
3. Then, whether the sequence number of the operation's request message is valid.
4. Finally, whether 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.

Prepare: This phase ensures that honest replicas/nodes in the network agree on the total order of requests within a view. Note that the pre-prepare and prepare phases together provide the total order to the messages. 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.

Commit: This phase ensures that honest replicas/nodes in the network agree on the total order of requests across views. 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.

In summary, the protocol works as follows:

1. A client sends a request to invoke a service operation in the primary.
2. The primary multicasts the request to the backups.
3. Replicas execute the request and send a reply to the client.
4. The client waits for replies from different replicas with the same result; this is the result of the operation.

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.

The normal view of the PBFT protocol can be visualized as follows:

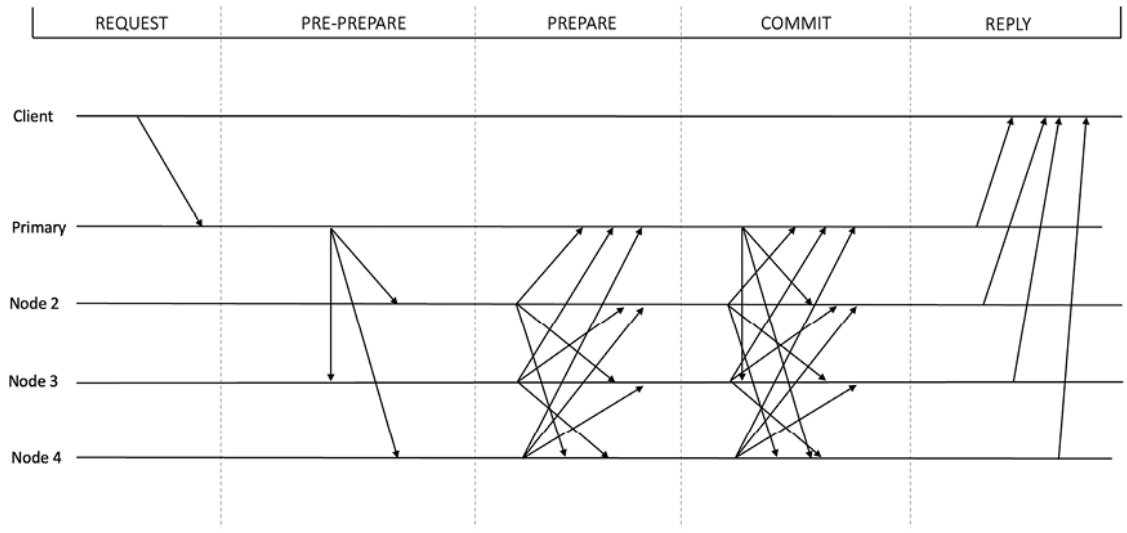

Figure 5.4: PBFT protocol

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 using digital signatures. In addition, certificates are used to ensure the adequate majority of participants (nodes).

> Do not confuse these certificates with digital certificates commonly used in **public key infrastructure** (**PKI**), or on websites and IT infrastructures to secure assets such as servers.

Certificates

Certificates in PBFT protocols are used to demonstrate that at least *2f+1* nodes have stored the required information. For example, if a node has collected *2f+1* prepare messages, 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. There are also a number of variables that the PBFT protocol maintains to execute the algorithm. These variables and their meanings are listed as follows:

State variable	Explanation
`v`	View number
`m`	Latest 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

We can now look at the types of messages and their formats, which becomes quite easy to understand if we refer to the preceding variables table.

Types of messages

The PBFT protocol works by exchanging several messages. A list of these messages is presented as follows with their format and direction.

The following table contains message types and relevant details:

Message	From	To	Format	Signed by
Request	Client	Primary	`<REQUEST, m>`	Client
Pre-prepare	Primary	Backups	`<PRE-PREPARE, v, n, h>`	Client
Prepare	Replica	Replicas	`<PREPARE, v, n, h, i>`	Replica
Commit	Replica	Replicas	`<COMMIT, v, n, h, i>`	Replica
Reply	Replica	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

Note that all these messages are signed. Let's look at some specific message types that are exchanged during the PBFT protocol.

View change occurs when a primary is suspected to be faulty. This phase is required to ensure protocol progress. With the view change subprotocol, 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 subprotocol.

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.

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.

The view change protocol can be visualized in the following diagram:

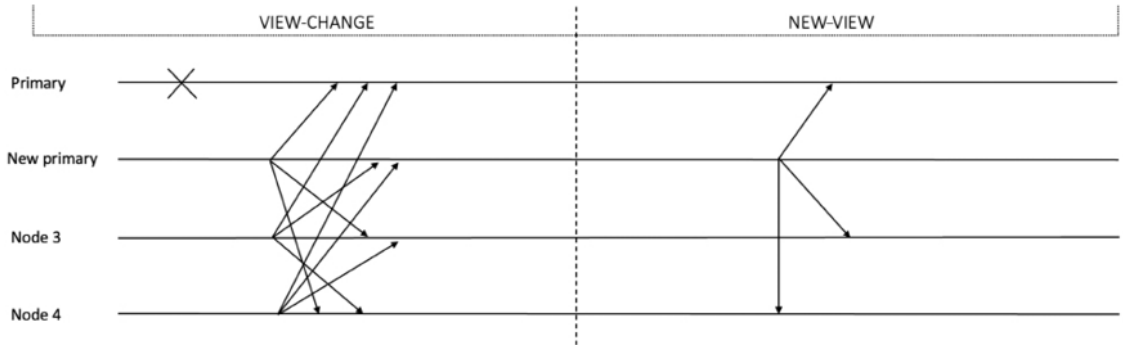

Figure 5.5: View change subprotocol

The view change subprotocol is a mechanism to achieve liveness. Three smart techniques are used in this subprotocol 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.

Checkpointing is another crucial subprotocol. 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 checkpoint in 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 is indeed a revolutionary protocol that has opened a new research field of PBFT protocols. The original PBFT does have many strengths, but it also has some limitations. We discuss most of the commonly cited strengths and limitations in the following table.

Strengths	Limitations
PBFT provides immediate and deterministic transaction finality. This is in contrast with the PoW protocol, where several confirmations are required to finalize a transaction with high probability.	Node scalability is quite low. Usually, smaller networks of few nodes (10s) are suitable for use with PBFT due to its high communication complexity. This is the reason it is more suitable for consortium networks instead of public blockchains.
PBFT is energy efficient as compared to PoW, which consumes a tremendous amount of electricity.	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 attack is not very practical in consortium networks because all identities are known on the network. This problem can

be addressed simply by increasing the number of nodes in the network.

In the traditional client-server model, PBFT works well; however, in the case of blockchain, directly implementing PBFT in its original state may not work correctly. This is because PBFT's original design was not developed for blockchain. This research resulted in IBFT and PBFT implementation in Hyperledger Sawtooth, Hyperledger Fabric, and other blockchains. In all these scenarios, some changes have been made in the core protocol to ensure that they're compatible with the blockchain environment.

More details on PBFT in Hyperledger Sawtooth can be found here: `https://github.com/hyperledger/sawtooth-rfcs/blob/master/text/0019-pbft-consensus.md`.

Istanbul Byzantine Fault Tolerance

IBFT was developed by AMIS Technologies as a variant of PBFT suitable for blockchain networks. It was presented in EIP 650 for the Ethereum blockchain.

Let's first discuss the primary differences between the PBFT and IBFT protocols. They are as follows:

- In IBFT, validator membership can be changed dynamically, and validators voted in and out as required. It is in contrast with the original PBFT, where the validator nodes are static.
- 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.
- 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 garbage collection in IBFT.

Now we've covered the main points of comparison between the two BFT algorithms, we'll examine how the IBFT protocol runs, and its various phases.

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 secure cryptography. The 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. The following flowchart shows how the IBFT protocol works:

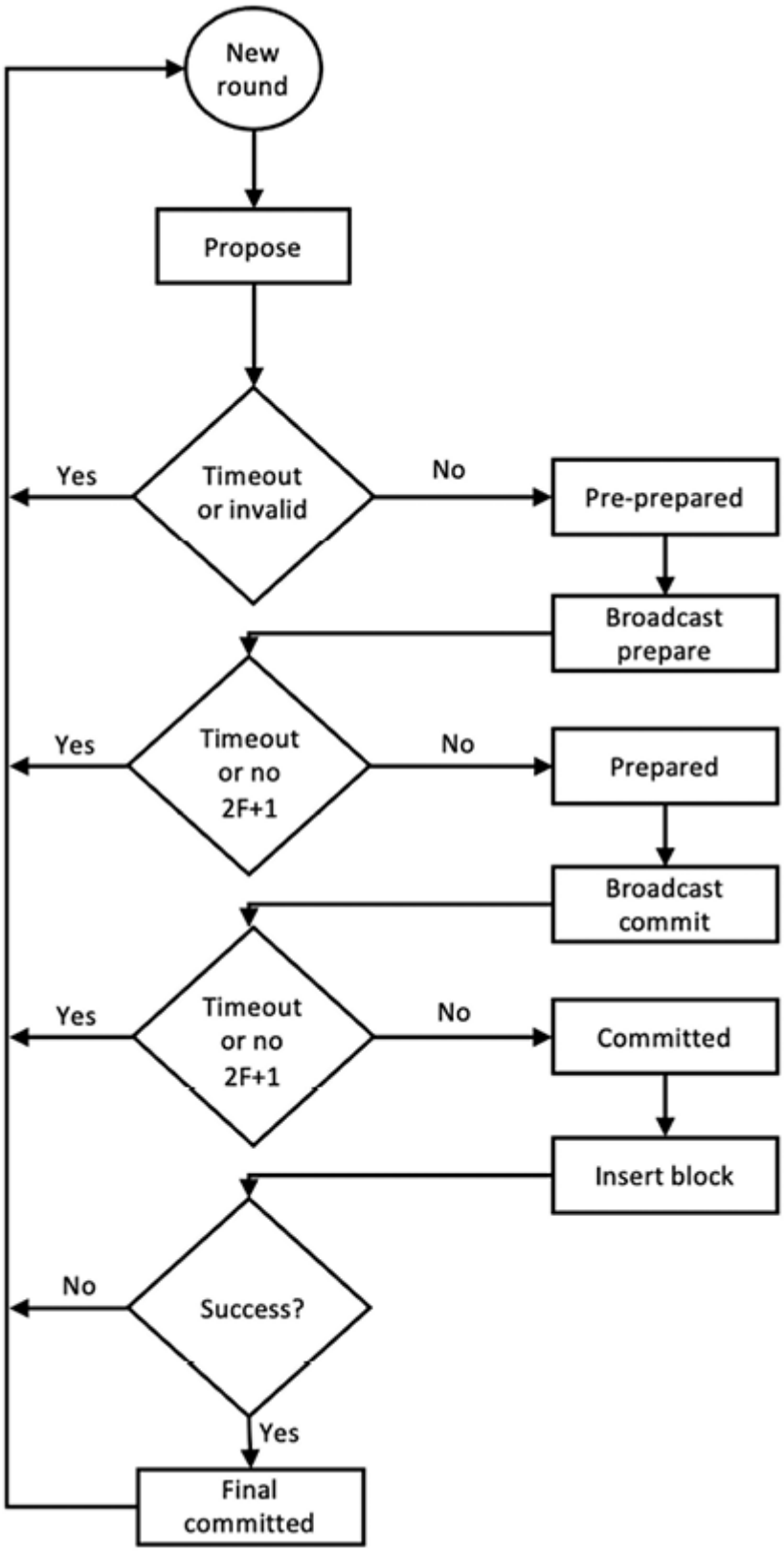

Figure 5.6: IBFT flowchart

We'll discuss this process step by step as follows:

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.
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, a 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.

Now that we've understood the flow of IBFT, let's now further explore which states it maintains and how.

Consensus states

IBFT is an SMR algorithm. Each validator maintains a state machine replica in order to reach block consensus, that is, agreement. These states are listed as follows with an explanation:

- **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.

- **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.

The IBFT protocol can be visualized using a diagram, similar to PBFT, as follows:

Figure 5.7: IBFT

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 been implemented in several blockchains. Sample implementations include Quorum and Celo. A Quorum implementation is available at the following link: **https://github.com/ConsenSys/quorum**.

IBFT, with some modifications, has also been implemented in the Celo blockchain, which is available at **https://github.com/celo-org/celo-blockchain**.

Several other consensus algorithms have been inspired by PBFT and have emerged as a result of deep interest in blockchain research. One such algorithm is Tendermint.

Tendermint

Tendermint is another variant of PBFT. It is inspired by both the DLS and PBFT protocols. Tendermint also makes use of the SMR approach to provide fault-tolerant replication. As we saw before, state machine replication is a mechanism that allows synchronization between replicas/nodes of the network.

> The DLS protocol is a consensus mechanism named after its authors (Dwork, Lynch, Stockmeyer). This protocol consists of two initial rounds. This process of rounds with appropriate message exchange ensures that agreement is eventually achieved on a proposed or default value. As long as the number of nodes in the system is more than *3f*, this protocol achieves consensus.

Traditionally, a consensus mechanism was used to run with a small number of participants, and thus performance and scalability were 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.

The Tendermint protocol 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 subprotocols for normal mode and view change mode (recovery in event of errors), Tendermint terminates without any additional communication costs.

We saw earlier, in the introduction, that each consensus model is studied, developed, and run under a system model with some assumptions about the system. Tendermint is also designed with a system model in mind. Now we'll define and introduce each element of the system model:

- **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. This means that the protocol operates correctly as long as the number of nodes in the network is more than $3f$, where f is the number of faulty nodes. This implies that, at a minimum, there have to be four nodes in a network to tolerate Byzantine faults.
- **Timing assumptions**: Under the network model, Tendermint assumes a partially synchronous network. This means that there is an unknown bound on the communication delay, but it only applies after an unknown instance of time called GST. Practically, this means that the network is eventually synchronous, meaning the network becomes synchronous after an unknown finite point in time.
- **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 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.

There are three fundamental consensus properties that Tendermint solved. As we discussed earlier in the chapter, generally in a consensus problem, there are safety and liveness properties that are required to be met. Similarly, in Tendermint, these safety and liveness properties consist 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()`.

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.

The following diagram depicts the Tendermint protocol at a high level:

Figure 5.8: Tendermint high-level overview

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 possible decision value is proposed, which has already been locked in a previous round. The proposal message also includes a value of a 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
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.
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.

We can visualize the protocol flow with the diagram shown here:

Figure 5.9: Tendermint flowchart

Types of messages

There are three types of messages in Tendermint:

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

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:

- `Timeout-propose`
- `Timeout-prevote`
- `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

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.

These variables are listed and discussed as follows:

- **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 $2f + 1$ pre-votes have already been received for this value, this is a possible decision value.
- **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.

Now we'll explore the new **termination** mechanism. For this purpose, there are two variables, namely `validValue` and `validRound`, which are used by the proposal message. Both 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, and they will 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.

> Tendermint is developed in Go. It can be implemented in various scenarios. The source code is available at https://github.com/tendermint/tendermint. It also is re-

leased as Tendermint Core, which is a language-agnostic programming middleware that takes a state transition machine and replicates it on many machines. Tendermint Core is available here: https://tendermint.com/core/.

The algorithms discussed so far are variants of traditional Byzantine fault-tolerant algorithms. Now, we'll introduce the protocols that are specifically developed for blockchain protocols.

Nakamoto consensus

Nakamoto consensus, or **PoW**, was first introduced with Bitcoin in 2009. Since then, it has stood the test of time and is the longest-running blockchain network. Contrary to common belief, the PoW mechanism is not a consensus algorithm but a Sybil attack defense mechanism. The consensus is achieved by applying the fork choice rule to choose the longest/heaviest chain to finalize the canonical chain. In this sense, we can say that PoW is a consensus facilitation algorithm that facilitates consensus and mitigates Sybil attacks, thus maintaining the blockchain's consistency and network's security.

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.

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.

> The original posting by Satoshi Nakamoto can be found here: https://satoshi.nakamotoinstitute.org/emails/cryptography/11/.

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.

In a nutshell, PoW works as follows:

- PoW makes use of hash puzzles.
- A node that proposes a block has to find a nonce such that H (*nonce* || *previous hash* || *Tx* || ... || *Tx*) < *Threshold value*. This threshold value or the target value is based on the mining difficulty of the network, which is set by adding or reducing the number of zeroes in front of the target hash value. More zeroes mean more difficulty and a smaller number of zeroes means less difficult to solve. If there are more miners in the network, then the mining difficulty increases, otherwise, it reduces as the number of miners reduces.

The process can be summarized as follows:

1. New transactions are broadcast to 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.
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 (the target 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.

We can visualize how PoW works with the diagram shown here:

Figure 5.10: PoW

Nakamoto versus traditional consensus

The Nakamoto consensus was the first of its kind. It solved the consensus problem, which had been traditionally solved using pre-Bitcoin protocols like PBFT. However, we can map the properties of PoW consensus to traditional Byzantine consensus.

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:

- 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 violations.
- 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.

> PoW chain quality and common prefix properties are introduced and discussed in the following paper:
>
> Garay, J., Kiayias, A. and Leonardos, N., 2015, April. *The bitcoin backbone protocol: Analysis and applications*. *In Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 281-310). Springer, Berlin, Heidelberg.

https://eprint.iacr.org/2014/765.pdf

Variants of PoW

There are two main variants of PoW algorithms, based on the type of hardware used for their processing:

- **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 Bitcoin PoW 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 toward 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** 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.

PoW consumes tremendous amounts of energy; as such, there are several alternatives suggested by researchers. One of the first alternatives proposed is PoS.

Alternatives to PoW

PoW does have various drawbacks, and the biggest of all is energy consumption—the total electricity consumed by Bitcoin miners is more than many countries! This is huge, and all that power is, in a way, wasted; no useful purpose is served except mining. Environmentalists have raised real concerns about this situation. In addition to electricity consumption, the carbon footprint is also very high, and is only expected to grow.

For recent data on Bitcoin's carbon footprint, visit

https://digiconomist.net/bitcoin-energy-consumption.

It has been proposed that PoW puzzles can be designed in such a way that they serve two purposes. First, their primary purpose is in consensus mechanisms, and second, they serve to perform some useful scientific computation. This way not only can the schemes be used in mining, but they can also help to solve other scientific problems.

Another drawback of the PoW scheme, especially the one used in Bitcoin (Double SHA-256), is that since the introduction of ASICs, the power is shifting toward miners or mining pools who can afford to operate large-scale ASIC farms. This power shift challenges the core philosophy of the decentralization of Bitcoin. There are a few alternatives that have been proposed, such as ASIC-resistant puzzles, which are designed in such a way that building ASICs for solving this puzzle is unfeasible and does not result in a major performance gain over commodity hardware. A common technique used for this purpose is to apply a class of computationally hard problems called **memory-hard computational puzzles**. The core idea behind this method is that as puzzle solving requires a large amount of memory, it is not feasible to be implemented on ASIC-based systems.

This technique was initially used in Litecoin and Tenebrix, where the Scrypt hash function was used as an ASIC-resistant PoW scheme. Even though this scheme was initially advertised as ASIC resistant, Scrypt ASICs became available a few years ago, disproving the original claim by Litecoin. This happened because even if Scrypt is a memory-intensive mechanism, initially, it was thought that building ASICs with large memories is difficult due to technical and cost limitations. This is no longer the case because memory hardware is increasingly becoming cheaper. Also, with the ability to produce nanometer-scale circuits, it is possible to build ASICs that can run the Scrypt algorithm.

Another approach to ASIC resistance is where multiple hash functions are required to be calculated to provide PoW. This is also called a **chained hashing scheme**. The rationale behind this idea is that designing multiple hash functions on an ASIC is not very feasible. The most common exam-

ple is the X11 memory hard function, implemented in Dash. X11 comprises 11 SHA-3 contestants, where one algorithm outputs the calculated hash to the next algorithm until all 11 algorithms are used in a sequence. These algorithms include BLAKE, BMW, Groestl, JH, Keccak, Skein, Luffa, CubeHash, SHAvite, SIMD, and ECHO. This approach did provide some resistance to ASIC development initially, but now ASIC miners are available commercially and support mining of X11 and similar schemes.

Perhaps another approach could be to design self-mutating puzzles that intelligently or randomly change the PoW scheme or its requirements as a function of time. This strategy will make it almost impossible to be implemented in ASICs as it will require multiple ASICs to be designed for each function. Also, randomly changing schemes would be practically impossible to handle in ASICs. At the moment, it is unclear how this can be achieved practically.

In the following sections, we consider some alternative consensus mechanisms to PoW.

Proof of Storage

Also known as **proof of retrievability**, this is another type of proof of useful work that requires storage of a large amount of data. Introduced by Microsoft Research, this scheme provides a useful benefit of distributing the storage of archival data. Miners are required to store a pseudo-randomly selected subset of large data to perform mining.

Proof of Stake

Proof of stake (**PoS**) is also called **virtual mining**. This is another type of mining puzzle that has been proposed as an alternative to traditional PoW schemes. It was first proposed in Peercoin in August 2012, and now, prominent blockchains such as EOS, NxT, Steem, Ethereum 2.0, Polkadot, and Tezos are using variants of PoS algorithms. Ethereum, with its Serenity release, will soon transition to a PoS-based consensus mechanism.

In this scheme, the idea is that users are required to demonstrate the possession of a certain amount of currency (coins), thus proving that they have a stake in the coin. The simplest form of this stake is where mining is made comparatively easier for those users who demonstrably own larger amounts of digital currency. The benefits of this scheme are twofold; first, acquiring large amounts of digital currency is relatively difficult as compared to buying high-end ASIC devices, and second, it results in saving computational resources. Various forms of stake have been proposed:

- **Proof of coinage**: The age of a coin is the time since the coins were last used or held. This is a different approach from the usual form of PoS, where mining is made easier for users who have the highest stake in the altcoin. In the coin-age-based approach, the age of the coin (coinage) is reset every time a block is mined. The miner is rewarded for holding and not spending coins for a period of time. This mechanism has been implemented in Peercoin combined with PoW in a creative way. The difficulty of mining puzzles (PoW) is inversely proportional to the coinage, meaning that if miners consume some coinage using coin-stake transactions, then the PoW requirements are relieved.
- **Proof of Deposit**: This is a type of PoS. The core idea behind this scheme is that newly minted blocks by miners are made unspendable for a certain period. More precisely, the coins get locked for a set number of blocks during the mining operation. The scheme works by allowing miners to perform mining at the cost of freezing a certain number of coins for some time.
- **Proof of Burn**: As an alternate expenditure to computing power, this method destroys a certain number of Bitcoins to get equivalent altcoins. This is commonly used when starting up a new coin projects as a means to provide a fair initial distribution. This can be considered an alternative mining scheme where the value of the new coins comes from the fact that, previously, a certain number of coins have been destroyed.

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. The chance of proposing the next block is directly proportional to the value staked by the participant.

In PoS systems, there is no concept of mining as 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.

PoS mechanisms generally select a stakeholder and grant appropriate rights to it based on its 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.

The diagram below shows how a generic PoS mechanism works:

Figure 5.11: PoS

In the preceding diagram, a stake calculator function is used to calculate the amount of staked funds, and based on that, a new proposer is selected. If for some reason the proposer selection fails, then the stake calculator and new proposer selection function run again to choose a new block proposer (leader).

Chain-based PoS 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* || *clock time*) < *target x stake value*

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 the 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.

> Peercoin was the first blockchain to implement PoS. Nxt (**https://www.jelurida.com/nxt**) and Peercoin (**https://www.peercoin.net**) are two examples of blockchains where chain-based PoS is implemented.

In **committee-based PoS**, 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. More details on this are available here: https://www.cardano.org/en/ouroboros/.

VRFs were introduced in *Chapter 4*, *Asymmetric Cryptography*. More information on VRF can be found here: https://datatracker.ietf.org/doc/html/draft-goldbe-vrf-01.

Delegated PoS 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 stake that participants have in 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.

Proof of Activity (PoA)

This scheme is a hybrid of PoW and PoS. In this scheme, blocks are initially produced using PoW, but then each block randomly assigns three stakeholders that are required to digitally sign it. The validity of subsequent blocks is dependent on the successful signing of previously randomly chosen blocks.

There is, however, a possible issue known of the **nothing at stake** problem, where it would be trivial to create a fork of the blockchain. This is possible because in PoW, appropriate computational resources are required to mine, whereas in PoS, there is no such requirement; as a result, an attacker can try to mine on multiple chains using the same coin.

Non-outsourceable puzzles

The key motivation behind this puzzle is to develop resistance against the development of mining pools. Mining pools, as previously discussed, offer

rewards to all participants in proportion to the computing power they consume. However, in this model, the mining pool operator is a central authority to whom all the rewards go and who can enforce specific rules. Also, in this model, all miners only trust each other because they are working toward a common goal, in the hope of the pool manager getting the reward. Non-outsourceable puzzles are a scheme that allows miners to claim rewards for themselves; consequently, pool formation becomes unlikely due to inherent mistrust between anonymous miners.

Next, we consider HotStuff, the latest class of BFT protocol with several optimizations.

HotStuff

HotStuff is the latest class of BFT protocol with several optimizations. There are several changes in HotStuff that make it a different and, in some ways, better protocol than traditional PBFT. HotStuff was introduced by VMware Research in 2018. Later, it was presented at the Symposium on Principles of Distributed Computing.

> The paper reference is as follows:
>
> Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G. and Abraham, I., 2019, July. HotStuff: *BFT consensus with linearity and responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (pp. 347-356). ACM.

The three key optimizations of HotStuff are listed below:

- **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 a 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.

- **Optimistic responsiveness** ensures that any correct leader after GST is reached only requires the first $n-f$ responses to ensure progress.
- **Chain quality** ensures fairness and liveness in the system by allowing fast and frequent leader rotation.

Another innovation in HotStuff is the separation of safety and liveness mechanisms. The separation of concerns allows for 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.

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

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.

However, note that there is one problem in this scheme where, if the leader becomes too overloaded, then the processing may become slow, impacting the whole network. Mesh and star topologies can be visualized in the following diagram:

Figure 5.12: Star topology

A second change is regarding PBFT's two main subprotocols, namely, normal mode and view change mode. In this model, the 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 subprotocol. Instead, the checking of the threshold of the messages to change the view becomes part of the normal view.

When compared with Tendermint, HotStuff is improved in two aspects. First, it can be chained, which means that it can be pipelined where a single quorum certificate can service in different phases at the same time, resulting in better performance. Second, it is optimistically responsive, as introduced above.

Just like PBFT, HotStuff also solves the SMR problem. Now, we'll describe how this protocol works. As we saw earlier, consensus algorithms are described and work under a system model.

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.

> We discussed threshold signatures in *Chapter 4*, *Asymmetric Cryptography*. You can refer to the details there if required.

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** (**QC**). 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.

- **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.
- **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.
- **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.
- **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.

This algorithm can be visualized in the following diagram:

Figure 5.13: HotStuff protocol

HotStuff, with some variations, is also used in DiemBFT, which is a distributed database designed to support a global currency proposed by Facebook.

More details on the Diem protocol are available here: https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf.

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 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 pro-

tocols. It is also a framework in which other protocols, such as DLS, PBFT, and Tendermint can be expressed.

In this section, we have explored various consensus algorithms in detail. We discussed pre-Bitcoin distributed consensus algorithms and post-Bitcoin, PoW-style algorithms. Now, a question arises naturally: with the availability of all these different algorithms, which algorithm is best, and which one should you choose for a particular use case? We'll try to answer this question in the next section.

Choosing an algorithm

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. We will discuss some of the main factors that can influence the choice of consensus algorithm. Note that these factors are different from the core safety and liveness properties discussed earlier, which are the fundamental requirements needed to be fulfilled by any consensus mechanism. Other factors that we are going to introduce here are use-case-specific and impact the choice of consensus algorithm. These factors include finality, speed, performance, and scalability.

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, 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 transac-

tion 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 a high 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 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 enterprise-grade performance requirements.

Summary

In this chapter, we explained some of the most prominent protocols in blockchain and traditional distributed system consensus. We covered several algorithms, including PoW, proof of stake, traditional BFT protocols, and the latest protocols, such as HotStuff. We did not cover newer algorithms such as BABE, GRANDPA, proof of history, and Casper which we will discuss later in this book.

Distributed consensus is a very interesting area of research, and academics and industry researchers are participating in this exciting subject. It is also a deep and vast area of research; therefore, it is impossible to cover every protocol and all dimensions of this gripping discipline in a single chapter. Nevertheless, the protocols and ideas presented in this chapter provide solid ground for further research and more in-depth exploration.

In the next chapter, we'll introduce Bitcoin, which is the first blockchain and cryptocurrency, invented in 2008.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://packt.link/ips2H