Chapter 2

Distributed Consensus Protocols and Algorithms

Yang Xiao[1], Ning Zhang[2], Jin Li[3], Wenjing Lou[1], Y. Thomas Hou[1]

[1]Virginia Polytechnic Institute and State University, USA

[2]Washington University in St. Louis, USA

[3]Guangzhou University, China

2.1 Introduction

Fault-tolerant consensus has been extensively studied in the context of distributed systems. By regulating the dissemination of information within the network of distributed components, a fault-tolerant consensus algorithm guarantees all components agree on common data values and perform the same course of actions in response to a service request, in spite of the presence of faulty components and unreliable communication links. This consensus guarantee is crucial to the normal functioning of a distributed system.

Being a realization of distributed system, a blockchain system relies on a consensus protocol for ensuring all nodes in the network agree on a single chain of transaction history, given the adverse influence of malfunctioning and malicious nodes. At the time of writing, there are over a thousand initiatives in the cryptocurrency plethora, embodying more than ten classes of consensus protocols. This chapter provides an overview of the basics of classic fault-tolerant consensus in distributed computing and introduces several popular blockchain consensus protocols.

We organize the chapter as follows: Section 2.2 introduces the basics of fault-tolerant consensus in distributed system and two practical consensus protocols for distributed computing. Section 2.3 presents the Nakamoto consensus protocol, a pioneering proof-of-work (PoW) based consensus protocol first used by Bitcoin. Section 2.4 presents several emerging non-PoW blockchain consensus protocols and their application scenarios. Section 2.5 gives a qualitative evaluation and comparison over the mentioned blockchain consensus protocols. Section 2.6 concludes this chapter and summarize the design philosophy for blockchain consensus protocols.

2.2 Fault-Tolerant Consensus in a Distributed System

In a distributed system, all components strive to achieve a common goal in spite of being separated geographically. Consensus, in the simplest form, means these components reach agreement on certain data values. In an actual system, the system components and their communication channels are prone to unpredictable faults and adversarial influence. In this section we discuss the consensus problem of message-passing systems[1] in the presence of two types of component failures: *crash failure* and *Byzantine failure*. We then study two practical consensus algorithms that tolerate these component failures in distributed computing. For convenience, the terms processor, node, and component are used interchangeably in this section.

2.2.1 The System Model

There are three major factors of consensus in a distributed system: network synchrony, component faults, and the consensus protocol.

Network Synchrony

Network synchrony is a basic concept in distributed system. It defines the degree of coordination of all system components. We need to assume a certain network synchrony condition before any protocol development or performance analysis. Specifically there are three network synchrony conditions:

- *Synchronous*: Operations of components are coordinated in rounds. This is often achieved by a centralized clock synchronization service. In each round, all components perform the same type of operations. For example,

[1]There is another type of distributed system called shared-memory system. Please refer to [1] for more details. In this chapter we adhere to message-passing system because of its resemblance to blockchain.

in round r all components broadcast messages to others and in round $(r+1)$ all components process the received messages and broadcast the outputs.

- *Asynchronous*: Operations of components are not coordinated at all. This is often the result of no clock synchronization service or the drifting effect of component clocks. Each component is not bound by any coordination rules and performs its own routine in an opportunistic fashion. There is no guarantee on message delivery or an upper bound of message transmission delay between components.

- *Partially synchronous*: Operations of components are not coordinated, but there is an upper bound of message transmission delay. In other words, message delivery is guaranteed, although may not be in a timely manner. This is the network condition assumed for most practical distributed systems.

In most application scenarios we assume the system is either synchronous or partially synchronous. For example, the voting process of a democratic congress is considered synchronous while the Bitcoin network is considered partially synchronous[2].

Faulty Component

A component is faulty if it suffers a failure that stops it from normal functioning. We consider two types of faulty behaviors that a component may suffer:

- **Crash Failure** The component abruptly stops functioning and does not resume. The other components can detect the crash and adjust their local decisions in time.

- **Byzantine Failure** The component acts arbitrarily with no absolute condition. It can send contradictory messages to the other components or simply remain silent. It may look normal from outside and not incur suspicion from others throughout the history of the network.

Byzantine failure got its name from Lamport, Shostak, and Pease's work on the Byzantine generals problem [2], which we will discuss later along with the Oral Messaging algorithm. A Byzantine failure is often the result of a

[2]Many research papers refer to Bitcoin network as "asynchronous". Since Bitcoin is based upon the Internet which guarantees message delivery, we follow the above taxonomy and consider Bitcoin network partially synchronous.

malfunctioning system process or the manipulation of malicious actor. When there are multiple Byzantine components in the network, they may collude to deal even more damage to the network. Byzantine failure is considered the worst case of component failures and crash failure is often seen as a benign case of Byzantine failure.

Consensus Protocol

A consensus protocol defines a set of rules for message passing and processing for all networked components to reach agreement on a common subject. A message passing rule regulates how a component broadcasts and relays messages while a processing rule defines how a component changes its internal state in response of received messages. As a rule of thumb, we say the consensus is reached when all no-faulty components agree on the same subject.

From security's perspective, the strength of a consensus protocol is usually measured by the number faulty components it can tolerate. Specially, if a consensus protocol can tolerate at least one crash failure, we call it crash-fault tolerant (CFT). Similarly, if a consensus protocol can tolerate at least one Byzantine failure, we call it Byzantine-fault tolerant (BFT). Because of the inclusive relationship between Byzantine failure and crash failure, a BFT consensus is naturally CFT. Moreover, consensus can never be guaranteed in an asynchronous network with even just one crash failure, let alone Byzantine failures. Interested readers are referred to [3] for the impossibility proof.

In the remainder of this chapter we focus on the Byzantine fault tolerance of consensus protocols in synchronous or partially synchronous networks.

2.2.2 Byzantine Fault Tolerant Consensus

Formally, we consider a distributed message-passing system with N components $C_1, C_2, ..., C_N$. Each component C_i has an input x_i and an output y_i that is not assigned until the first round of consensus execution. Components are inter-connected by communication links that deliver output messages across the network.

Consensus Goal The BFT consensus for the above system must satisfy the following conditions [4]:

- **Termination:** Every non-faulty component decides an output.
- **Agreement:** All non-faulty components eventually decide the same output $\hat{y}$.

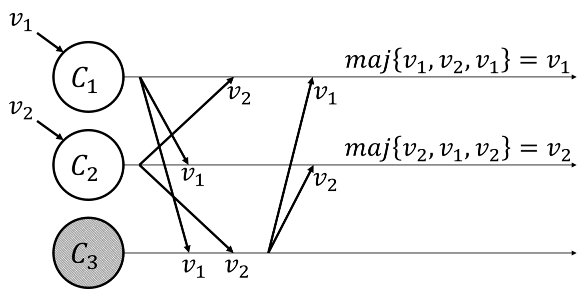

Figure 2.1: Example for Theorem 1: a three-component message-passing system with one component being Byzantine.

- **Validity:** If all components begin with the same input $\hat{x}$, then $\hat{y} = \hat{x}$.
- **Integrity:** Every non-faulty component's decision and eventually $\hat{y}$ must have been proposed by some non-faulty component.

The integrity condition ensures that the consensus result $\hat{y}$ should not originate from an adversary. In many older textbooks and research papers it is often not included for the reason that the origin of $\hat{y}$ is not important, as long as $\hat{y}$ is a legal result of the consensus process (validity) and accepted by all non-faulty components (agreement). Here we value a correct origin of the consensus result and consider integrity as an essential part of the consensus goal.

For an algorithm to achieve BFT consensus, the super majority (more than two thirds) of the components must be non-faulty. A more precise statement is given in Theorem 1.

Theorem 1. *In a message-passing system with n components, if f components are Byzantine and $n \leq 3f$, then it is impossible for the system to reach the consensus goal.*

Theorem 1 can be conveniently proved by contradiction in a scenario components are partitioned into three groups, with one group consisting of all the Byzantine components. Interested reader may refer to [5, 6, 1] for different flavors of proofs, all of which are based on the partitioning scheme.

To better illustrate Theorem 1, a three-component system example is shown in Figure 2.1. In this system component C_1, C_2 are honest while component C_3 is Byzantine. All input/decision values are taken from the bivalent set $\{v_0, v_1\}$. Assume the initial input values for C_1 and C_2 are v_1 and v_2 respectively, and the consensus algorithm is as simple as choosing the majority value of all values received. After C_1, C_2 broadcast their values, C_3 sends v_1 to C_1 and v_2 to C_2. As a result, C_1 decides v_1 while C_2 decides v_2, violating the agreement condition of the consensus goal. Therefore in order to tolerate one Byzantine components, the network size should be at least four. In the general case, for any distributed system with N components and f being Byzantine, $N \geq 3f + 1$ is required to ensure consensus.

2.2.3 The OM Algorithm

First we describe the Byzantine generals problem. N Byzantine generals, each commanding an equal-sized army, have encircled an enemy city. They are geographically separated and can communicate only through messengers. To break the stalemate situation, each general votes to attack or retreat by sending messengers to other generals. Each general makes his/her decision locally based on the votes received. To complicate the situation, there are traitors within the generals who will sabotage the consensus by sending contradicting votes to different generals. The ultimate goal is for all loyal generals to agree on the same action, as halfhearted attack or retreat will result in debacle.

The Oral Messaging algorithm (OM) was proposed as a solution in the original Byzantine generals problem paper [2]. It assumes within the N generals there is a "commander" who starts the algorithm and the other $N - 1$ called "lieutenants" who orally pass around messages they received. The network is synchronous and the protocol proceeds in rounds. The pseudo code is shown in Algorithm 1 and Algorithm 2. Specially, we assume the commander knows at most f generals will be faulty (including him/herself) and starts the consensus process by executing the *OM(f)* algorithm. Note that DEFAULT is a predetermined value, either "retreat" or "attack".

Algorithm 1: $OM(f), f > 0$

```
1  Commander sends its value to every lieutenant;
2  for $i$ = 1 : N-1 do
3      Lieutenant $i$ stores the value received from Commander as $v_{i,i}$;
         $v_{i,i}$ =DEFAULT if no value received;
4      Lieutenant $i$ performs $OM(f-1)$ as Commander to send the value $v_{i,i}$ to
         the other $N-2$ lieutenants;
5  end
6  for $i$ = 1 : N-1 do
7      for $j$ = 1 : N-1 and $j \neq i$ do
8          Lieutenant $i$ stores the value received from lieutenant $j$ as $v_{i,j}$;
             $v_{i,j}$ =DEFAULT if no value received;
9      end
10     Lieutenant $i$ uses $majority\{v_{i,1}, v_{i,2}, ..., v_{i,N-1}\}$;
11 end
```

Algorithm 2: $OM(0)$ (The base case for $OM(f)$)

```
1 Commander sends its value to every lieutenant;
2 for $i = 1 : N-1$ do
3     Lieutenant $i$ stores the value received from Commander as $v_{i,i}$;
        $v_{i,i}$ =DEFAULT if no value received;
4     Lieutenant $i$ uses $v_{i,i}$;
5 end
```

Since the oral messaging algorithm is executed in a recursive fashion in which the recursion ends at $OM(0)$, it requires $f+1$ rounds of executions. Essentially, as long as $N \geq 3f+1$, the $f+1$ rounds of recursive executions guarantee that at the end of algorithm every general has exactly the same set of votes, from which the *majority* function then produces the same result — the consensus is achieved. Due to its recursive fashion, $OM(f)$ algorithm has $O(N^{f+1})$ message complexity, which is impractical when N is large.

2.2.4 Practical Consensus Protocols in Distributed Computing

Now we have discussed single-value consensus in a synchronous network. In a typically distributed computing system, the clients spontaneously issue computing requests while the distributed servers work as a consortium to provide correct and reliable computing service in response to these requests. The correctness requirement means not only every single request should be processed correctly, but also the sequence of requests from a client (or a group of clients) should be processed in the correct order, which is called the total ordering requirement. The combination of the two requirements makes distributed computing a significantly harder task than the single-value consensus problem we have seen so far. Moreover, the asynchronous nature of real-world networks further complicates the problem. In practice, we assume the real-world distributed computing network is partially synchronous with bounded communication delay between two non-faulty servers.

Replication In actual distributed computing systems, replication is the de facto choice for ensuring the availability of the system and the integrity of the service in face of faulty servers. A replication-based distributed system maintains a number of redundant servers in case the primary server crashes or malfunctions. The redundant servers are also called *backups* or *replicas*. There are two major types of replication schemes: primary-backup and state-machine replication.

- **Primary Backup (PB)** PB is also known as passive replication. It was first proposed in [7]. In a PB based system of n replicas, one replica is designated as the primary and the others are backups. The primary interacts with clients and processes clients' operation requests. After the primary finishes one task, it sends to the backups what it has done. If the primary crashes, a replica will be selected to assume the role of the primary. PB only tolerates crash failures; it does not tolerate any number of Byzantine failures.

- **State-Machine Replication (SMR)** SMR is also known as active replication. It was proposed in [8]. In a SMR based system, the consensus protocol is instantiated at each server which runs a deterministic state machine that receives inputs, changes states and produces outputs in an "organized" manner. This enables the distributed network to provide fault-tolerant service by replicating the state machine across server replicas and processing clients' operation requests in a coordinated way. A good SMR protocol should guarantee two basic service requirements: *safety* - all processors execute the same sequence of requests, and *liveness* - all valid requests are executed.

Next we introduce two well-known SMR based consensus protocols for distributed computing: **Viewstamped Replication** and **Practical Byzantine Fault Tolerance**.

Viewstamped Replication (VSR)

VSR is an early protocol developed for distributed replication systems. Here we present an updated version of VSR proposed by Liskov and Cowling in 2012 [9]. Interested reader may refer to [10] for Oki and Liskov's original design. In a VSR system with N replicas, there is one *primary* and $N-1$ backups. Each replica operates a local state machine with state variables listed in Table 2.1. The "viewstamp" refers to the $\langle v, n\rangle$ pair, which essentially enables the replication network to process clients' operation requests in the correct order.

VSR consists of three sub-protocols; each is designed specially for one of the three status cases. The messages involved are listed in Table 2.2. We will leave out the message details and focus on the high-level work flow of these protocols.

1) Normal operation protocol The normal operation runs from session to session when all functioning replicas hold the same view and the primary is in good condition. A session includes the client sending request and the replicas processing this request. A diagram of the normal operation protocol for a three-replica system is shown in Figure 2.2. At the beginning of a session,

Table 2.1: State variables at replica i in VSR

Variable	**Description**
i	self index
rep-list	list of all replicas in the network
status	operation status: either NORMAL, VIEW-CHANGE, or RECOVERING
v	current view number
m	the most recent request message from a client
n	sequence number of m
e	$=$ EXECUTE(m), the execution result of m
c	sequence number of the most recently committed client request
log	record of operation requests received so far
client-table	record of most recent operation for all clients

Table 2.2: Messages in VSR

Message	**From**	**To**	**Format**
Request	client	primary	$\langle \text{REQUEST}, m \rangle$
Prepare	primary	all backups	$\langle \text{PREPARE}, m, v, n, c \rangle$
PrepareOK	replica i	primary	$\langle \text{PREPAREOK}, v, i \rangle$
Reply	primary	client	$\langle \text{REPLY}, v, e \rangle$
Commit	primary	all backups	$\langle \text{COMMIT}, v, c \rangle$
StartViewChange	replica i	all replicas	$\langle \text{STARTVIEWCHANGE}, v+1, i \rangle$
DoTheViewChange	replica i	new primary	$\langle \text{DOTHEVIEWCHANGE}, v+1, i \rangle$
StartView	primary	all replicas	$\langle \text{STARTVIEW}, v+1, log \rangle$
Recovery	replica i	all replicas	$\langle \text{RECOVERY}, i \rangle$
RecoveryResponse	replica i	the recoverer	$\langle \text{RECOVERYRESPONSE}, v, n, c, i \rangle$

the client sends to the primary a *Request* message indicating a new operation request.

1. **Prepare** Upon receiving a request message, the primary updates its n, *log*, *client-table* and then passes this request to all backups using *Prepare* messages which also include its n and c which was updated in the previous session. Each backup executes the primary-committed operations if there is any and updates its state accordingly.

2. **PrepareOK** Each backup sends a *PrepareOK* message to the primary showing its state is up to date. After receiving f *PrepareOK* messages, the primary executes the requested operation and then updates c, *log*, *client-table*.

The primary then sends a *Reply* message to the client. Specifically if the primary hasn't received a client request for a long time, it sends a *Commit* message to the backups indicating the updated c, as an alternative to the *Prepare* message.

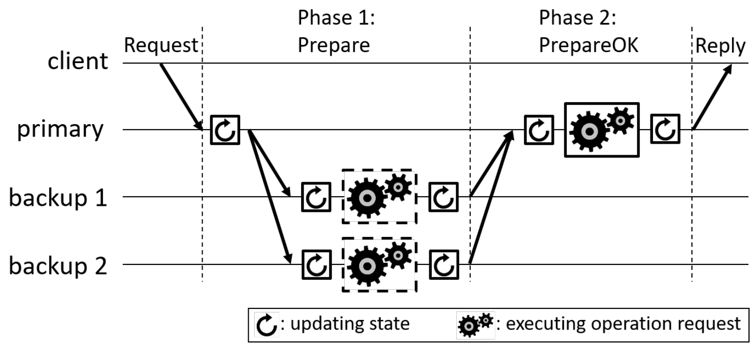

Figure 2.2: The normal operation protocol of VSR for a three-replica system.

2) View change protocol A view change is needed in the event of primary failure, which can be detected by a backup replica if no *Prepare* or *Commit* message has been received for a long time. After detecting the need for a view change, a replica updates its status to VIEW-CHANGE and advances the view number to $v + 1$. It then sends a *StartViewChange* message including the new view number $v + 1$ to other replicas. When a replica receives at least f *StartViewChange* messages with the new view number $v + 1$, it sends a *DoTheViewChange* message to the backup that will become the primary. When the new primary receives at least $f + 1$ *DoTheViewChange* messages, it updates its state accordingly, sends to other replicas a *StartView* message with the updated *log* and the new view number $v + 1$, and starts processing operation requests from clients. In the meantime, backup replicas receive the *StartView* message and update their state according to the *log* in the message, and finally change status to *normal.*

3) Recovering protocol When a replica recovers from a crash, it has to go through the recovering protocol before participating in normal operation and view change. It first sends a *Recovery* message to all other replicas. Each replica responds with a *RecoveryResponse* message indicating the current v. The primary needs to respond with additional state information including *log*, n, and c. The recovering replica waits until it has received at lease $f + 1$ *RecoveryResponse* messages, and then updates its state accordingly.

Fault Tolerance Note VSR can tolerate f crash failures if the total number of replicas (including the primary) $N \geq 2f + 1$. However, it has zero tolerance of Byzantine failures. For example if the primary is Byzantine faulty due to the adversarial manipulation, it can simply deny all client operation requests while pretending to work normally with the backups. If a backup is Byzantine on the other hand, it may maliciously initiate a view change session to oust the current primary.

Complexity analysis We analyze the message complexity of the normal operation. The communication overhead is primarily contributed by the two phases: the Prepare phase the primary broadcasts a *Prepare* message to all replicas; and the PrepareOK phase all replicas send a *PrepareOK* message to the primary. Therefore the message complexity for VSR's normal operation is $O(N)$.

Practical Byzantine Fault Tolerance (PBFT)

In the practical scenario where the distributed computing system may be compromised by malicious actors, both the primary and the backups are subject to adversarial manipulation, which falls into the realm of Byzantine failures. Proposed by Castro and Liskov in 1999 [11], PBFT advances VSR for tolerating Byzantine failures.

PBFT consists of three sub-protocols: normal operation, checkpoint, and view-change. The state variables at a replica are listed in Table 2.3 and the messages involved are listed in Table 2.4. As an additional security measure, each message is signed by the sender and verified by the receiver. In the

Table 2.3: State variables at replica i in PBFT

Variable	**Description**
i	self index (0 for primary)
rep-list	list of all replicas in the network
σ_i	replica i's key for signing messages
status	operation status: NORMAL or VIEW-CHANGE
v	current view number
m	the most recent request message from a client
n	sequence number of m
d	$=$ DIGEST(m), the digest of m
e	$=$ EXECUTE(m), the execution result of m
s	The latest checkpoint
h	low-water mark, ie. sequence number of s
H	high-water mark; $\langle h, H\rangle$ form a sliding window of length K.
$\mathcal{C}$	set of all valid *Checkpoint* messages proving the correctness of s
$\mathcal{P}_t$	set of a valid *Pre-prepare* message and all matching *Prepare* messages for a request with sequence number t
$\mathcal{P}$	set of the $\mathcal{P}_t$ for every request t that is higher than n
$\mathcal{V}$	set of all valid *View-Change* and *View-Change* messages
$\mathcal{O}$	set of a specially chosen *Pre-Prepare* messages
log	record of operation requests received so far

Table 2.4: Messages in PBFT

Message	**From**	**To**	**Format (signed)**
Request	client	primary	$\langle \text{REQUEST}, m \rangle_{\sigma_c}$
Pre-Prepare	primary	all backups	$\langle \text{PRE-PREPARE}, v, n, d \rangle_{\sigma_0}$
Prepare	replica i	all replicas	$\langle \text{PREPARE}, v, n, d, i \rangle_{\sigma_i}$
Commit	replica i	all replicas	$\langle \text{COMMIT}, v, n, d, i \rangle_{\sigma_i}$
Reply	replica i	client	$\langle \text{REPLY}, e, i \rangle_{\sigma_i}$
View-Change	replica i	all replicas	$\langle \text{VIEW-CHANGE}, v+1, n, \mathcal{C}, \mathcal{P}, i \rangle_{\sigma_i}$
New-View	primary	all replicas	$\langle \text{NEW-VIEW}, v+1, \mathcal{V}, \mathcal{O} \rangle_{\sigma_0}$
Checkpoint	replica i	all replicas	$\langle \text{CHECKPOINT}, n, d, i \rangle_{\sigma_i}$

following part we assume there are at most f faulty replicas and the network size $N = 3f + 1$. Later we will show $N \geq 3f + 1$ guarantees the protocol's Byzantine fault tolerance.

1) Normal operation protocol Similar to VSR, PBFT runs its normal operation from session to session. A diagram of normal operation for a four-replica system is shown in Figure 2.3. A session starts with a client operation request and goes through three sequential phases of replica interaction, namely Pre-Prepare, Prepare, and Commit, before replying to the client.

1. **Pre-Prepare** When the primary receives an operation request message m, it assigns a sequence number n to the request and sends a *Pre-Prepare* message along with the message m to all backups. After receiving a *Pre-Prepare* message, a backup checks the associated signatures and the validity of v, n, d. If everything is valid and n is within the water marked range $\langle h, H \rangle$, the backup accepts this message, updates its state accordingly, and proceeds to the Prepare phase.

2. **Prepare** Each backup sends a *Prepare* message to all other replicas. A replica that has received at least $2f + 1$ *Prepare* messages with the same v, n, d values updates its state accordingly and proceeds to the Commit phase.

3. **Commit** Each replica sends a *Commit* message to all other replicas. When a replica receives at least $2f + 1$ *Commit* messages with the same v, n, d values, it first finishes executing the old requests with sequence numbers lower than n, then executes the current request m to produce the result e, and finally updates its state accordingly.

When a replica finishes the Commit phase, it sends the execution result e in a *Reply* message to the client. The client accepts an execution result only after it receives at least $2f + 1$ *Reply* messages containing the same result e.

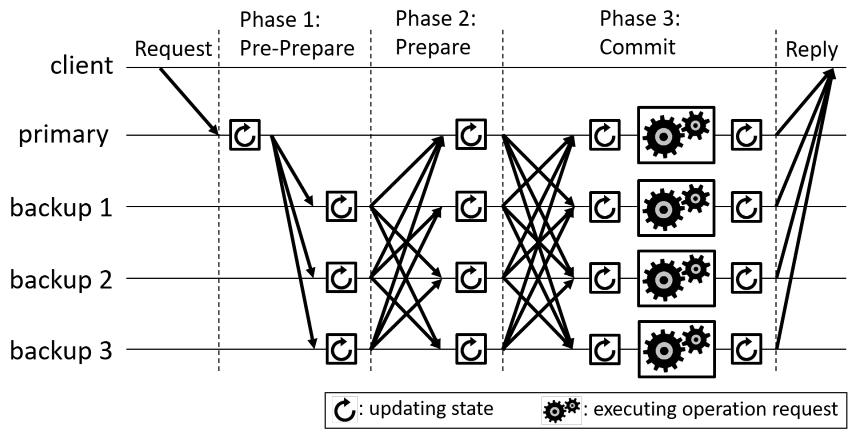

Figure 2.3: The normal operation protocol of PBFT for a four-replica system.

2) Checkpoint protocol The checkpoint protocol is used by the replicas to safely discard old items in *log* and agree on a stable checkpoint which provides essential service state info for the view-change process. Each replica periodically marks a executed client request as a checkpoint in *log* and records its sequence number as h, which is called the low water mark. It multicasts the checkpoint to other replicas in the form of a *Checkpoint* message. When a replica collects at least $2f + 1$ *Checkpoint* messages with the same n and d, it marks this checkpoint *stable* by assigning n to the variable h, and saves these *Checkpoint* messages as the proof from the stable checkpoint. After that the replica can safely discard from its *log* all *Pre-Prepare*, *Prepare*, and *Commit* messages with sequence numbers prior to h. In addition to h, each replica also updates the high water mark H so that the pair $\langle h, H \rangle$ form sliding window of length K. Note K is user-defined..

3) View-change protocol Since a view is bound to a known primary, when the primary is suspected faulty, the backups carry out the view-change protocol to choose a new primary. When a backup received a request but has not executed it for a certain timeout (for example it stays in Phase2 of normal operation for too long), it stops receiving further messages related to the current view v and updates status to VIEW-CHANGE before sending a *View-Change* message for view $v + 1$ to all replicas. When the new primary receives at least $2f$ *View-Change* messages for view $v + 1$, it multicasts a *New-View* message to all backups, updates its *log* and $\langle h, H \rangle$ pair, and proceeds into normal operation. A replicas validates the received *New-View* message, updates it state, and proceeds to normal operation as well.

Fault tolerance In the normal operation, the separation of pre-prepare phase and prepare phase is essential to the correct ordering of request execution and faulty primary detection. When a primary sends a *Pre-Prepare* messahe with out-of-order request or stays silent for a long time, the backups will consider the primary faulty and initiate the view change protocol for a new primary, as long as the majority of backups are non-faulty. Now we discuss the condition for PBFT to tolerate f Byzantine replicas. In the normal operation, a replica needs to receive $2f + 1$ *Prepare* messages with the same state to proceed to the Commit phase; it then needs to receive $2f + 1$ *Commit* messages with the same state to proceed to request execution. This is equivalent to the scenario we discussed in the Oral Messaging algorithm for the Byzantine generals problem: in a fully connected network the consensus can be reached if more than two thirds of components are non-faulty. The same consensus routine is applied in the checkpoint protocol and view-change protocol as well to guarantee the safety of the new primary election. As we have assumed $N = 3f + 1$ in the beginning, messages from $2f + 1$ non-faulty replicas are just enough to tolerate f Byzantine replicas. In the general case where f is unknown (but $N \geq 3f + 1$ is assumed), this number should be updated to $\lfloor \frac{2N}{3} \rfloor + 1$ from $2f + 1$ in the protocol to tolerate at least f Byzantine failures.

Complexity analysis We analyze the message complexity of the normal operation. The communication overhead is primarily contributed by the three phases: in the Pre-Prepare phase the primary broadcasts a message to all backups ($O(N)$); in the Prepare phase every backup broadcasts a message to all other replicas ($O(N^2)$); in the Commit phase every replica broadcasts a message to all other replicas ($O(N^2)$). Therefore the overall message complexity of PBFT's normal operation is $O(N^2)$. This is acceptable for a network that is fully or near-fully connected, unless N is huge.

Comparison between VSR and PBFT

VSR and PBFT are compared in Table 2.5. To summarize, PBFT achieves Byzantine fault tolerance with a more complex protocol scheme and higher communication overhead. To date PBFT has gained considerable interest in the blockchain community for its application in blockchains with small network size and permissioned access. We will introduce it in Section 2.4.

2.3 The Nakamoto Consensus

Since its inception in 2008, Bitcoin has become the leading figure in the cryptocurrency plethora. As of the first quarter of 2018, the Bitcoin network has

Table 2.5: A Comparison between VSR and PBFT for partially asynchronous distributed computing systems

	VSR	**PBFT**
Year proposed	1988	1999
CFT condition	$N \geq 2f + 1$	$N \geq 2f + 1$
BFT condition	Not supported	$N \geq 3f + 1$
Message Complexity	$O(N)$	$O(N^2)$

about 10,000 mining nodes and market capitalization of more than 100 billion dollars. The popularity of Bitcoin and other cryptocurrencies has brought huge academic and industrial interest to blockchain, the enabling technology behind the cryptocurrencies and many emerging distributed ledger systems.

Out of various aspects of Bitcoin, the celebrated Nakamoto consensus [12] is the key innovation to its security and performance. Similar to distributed computing systems, the consensus target for blockchain is the network's entire transaction history - not only the transactions' content, but also the their chronological order. In a practical blockchain system such as Bitcoin and Ethereum, the consensus protocol also needs to consider various physical factors such as network connectivity, network size, and adversarial influence. In this section we introduce the Nakamoto consensus protocol from a distributed system point of view.

2.3.1 The Consensus Problem

Consensus Goal The goal of the Nakamoto consensus is all nodes form a unified view on the network's transaction history. Similar to the four conditions for BFT consensus in the previous section, here gives the adapted conditions for the Nakamoto consensus:

- **Finality (Probabilistic):** For every block that has been attached to the blockchain, its drop-off probability asymptotically decreases to zero.
- **Agreement:** Every block is either accepted or dropped off by all honest nodes. If accepted, it should have the same block number in all blockchain replicas. In other words, all honest nodes agree on the same blockchain.
- **Validity:** If all nodes receive a same valid block, then this block should be accepted into the blockchain. The genesis block is good example.
- **Hash-Chain Integrity:** The blockchain contains all blocks up to the current block number. For block B with block number t and block B'

with block number $t + 1$, the hash value of the previous block in B' is the hash of B.

2.3.2 Network Model

Like most public blockchain networks, the Bitcoin network is a peer-to-peer overlay network based upon the Internet. Every node runs an instance of the Nakamoto protocol and maintains a replica of the blockchain. We model the network as a partially synchronous message-passing system with bounded transmission delay between two honest nodes, the same network model we assumed for the distributed computing systems in section 2.2. In addition to network synchrony, the Bitcoin network also features permissionless access and gossip-fashion information propagation.

Permissionless Access

Bitcoin is the first permissionless blockchain system and no authentication is required for a new player to instantiate a node and participate in the network. Specifically, to join the network a fresh node needs to set up in three steps:

1. Fetch a list of initial peers from several known DNS servers.
2. Search for new peers by asking its current peers and listening for spontaneous advertisements from other peers. Make sure the number of peers does not go below a minimum value (currently 8 for Bitcoin).
3. Retrieve a blockchain replica from peers and start normal operation.

To leave the network, a node simply disconnects. It will be gradually purged from the peer-lists of its peers. Since the transactions containing the node's public addresses have been written in the blockchain, the node can reclaim the same public address and hence its unspent transaction outputs when it rejoins the network using the same private key.

Information Propagation

The information propagation and message passing dynamics were first analyzed in [13]. There are two types of messages contributing to the consensus process: *transaction* and *block*. Figure 2.4 shows the diagram of one-hop propagation of a block. The propagation of transactions is in the same manner. The *validation* of a block consists of the validation of all transactions in the block and the verification of the hash value of the block header. The *advertise* message contains the hash of the validated block (or a list hashes of validated blocks).

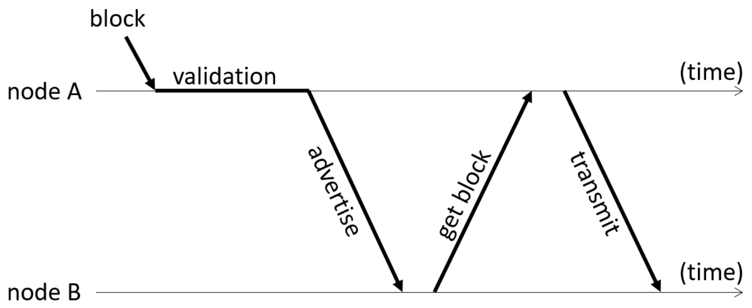

Figure 2.4: One-hop block propagation between two nodes

If node B sees a block that is new to its blockchain replica, it sends to node A a *get block* message containing the hash of the desired block. Finally node A *transmit* the desired block to node B, which then repeats this process with its own peers except node A.

Note that once a node B has an economic incentive to pass the block around. When other nodes know this block, they are less likely to create conflicting blocks (which will cause a fork) and more likely to accept the later block created by B, which eventually helps B make better use of its computation power and harvest more block benefits.

2.3.3 The Consensus Protocol

The Nakamoto consensus protocol is executed in a distributed fashion. Each node runs the same protocol and manages its own blockchain replica independently. The security of the consensus depends on the majority of nodes being honest, ie. running the correct version of the Nakamoto protocol. The protocol can be summarized into the following four rules for a single node:

1. **Message Passing Rule:** All newly received or locally generated blocks and transactions should be broadcast to peers in a timely manner.

2. **Validation Rule:** Blocks and transactions need to be validated before being broadcast to peers or appended to the blockchain. Invalid blocks and transaction are discarded.

3. **Longest-Chain Rule:** The longest chain is always the desired chain. Mining should be aimed at extending the longest chain by appending new blocks to it. If the node receives a valid block B^* with the same height as the block B that it is still working on, it discards B and appends the B^* to the blockchain and starts working on the new chain.

4. **Proof-of-Work (PoW):** The generation of a block includes inserting a nonce into the block header. The hash of the header should be less than a particular value, which is also called the PoW difficulty. Higher PoW difficulty yields more hashing operations expected for finding such a nonce. For security reasons, the PoW difficulty is automatically adjusted so that the average block generation interval of the network remains a constant value as the gross hashing power fluctuates (currently 10 minutes for Bitcoin).

As a result, the majority decision of the network is represented by the longest chain, which embodies the greatest amount of PoW computation effort.

Probabilistic finality According to the longest-chain rule, blocks that end up in a chain branch that is not the suffix of the longest chain shall be discarded or "orphaned". This means any block in the blockchain (except the genesis block) can be revoked, since it is possible for a powerful attacker to start from an early block and redo the proof-of-works all the way up to current blockchain height so that the network will acknowledge this new chain as the longest. On the bright size, if the attacker has less than 50% of the network's gross hashing power, it will produce blocks slower than the rest of the network. Let p denote the hashing power percentage controlled by the attacker and $p < 50\%$. Then the probability that the attacker will eventually catch up from behind m blocks is:

$$P\{\text{Catch-up}\} = \big(\frac{p}{1-p}\big)^m \tag{2.1}$$

Since $p < 50\%$, this probability drops exponentially as m increases. In other words, revoking such block from the blockchain is computationally impossible if more than half of the hashing power is owned by honest nodes and m is large. Currently in Bitcoin $m = 6$ is used as the transaction confirmation time. All blocks that have at least 6 descendants are considered probabilistically finalized.

2.4 Emerging Blockchain Consensus Algorithms

Due to the inherently tight trade-off between security and scalability in PoW based blockchains, researchers and developers have been exploring new blockchain schemes to support higher transaction volume and larger network size with lower energy consumption. This section introduces several promising non-PoW consensus algorithms: proof-of-stake (PoS), PBFT-based consensus protocols, Ripple consensus protocol, proof-of-elapsed-time (PoET). These algorithms are proposed either as alternatives to PoW for public blockchains (PoS, PoET) or for domain-specific applications (PBFT-based, Ripple). We will go through their consensus protocols and briefly analyze their fault-tolerance limits and security concerns.

2.4.1 Proof-of-Stake

Proof-of-Stake (PoS) was proposed by the Bitcoin community as an alternative to PoW. Compared to PoW in which miners race for the next block with brute computing force, PoS resembles a new philosophy of blockchain design, according to which the race should be carried out in a "civilized" manner that saves energy and maintains security. PoS maintains a set of validators who participate in the block generation process by depositing an amount of currency (stake) in the competition, which is designed in a way that a bigger stakeholder has a higher chance to win the competition.

There two major types of PoS: *Chain-based* PoS and *BFT-style* PoS. Chain-based PoS is the original design of PoS and got its name from retaining the longest-chain rule of Nakamoto consensus. It was first implemented in the cryptocurrency Ppcoin, later known as Peercoin [14]. In comparison, BFT-style PoS leverages the established results of BFT consensus for finalizing new blocks. In this section we introduce the basics of chain-based PoS. BFT-style PoS will be introduced along with other BFT consensus protocols in 2.4.2.

Chain-based PoS

In chain-based PoS, the blockchain maintains a set of validators who participate in the competition for the right to generate the next block. For every block generation cycle, chain-based PoS runs in two steps:

- **Step 1.** Every validator invests a stake in the competition for block generation. The deposited stakes are kept frozen until the end of this block generation cycle.
- **Step 2.** After a validator deposits its stake, it starts to generate new blocks similar to Nakamoto's proof of work, but with a limited difficulty which is further discounted by its stake value. The first block produced is immediately appended to the longest chain and the corresponding validator claims the block reward.

Fault tolerance Analogous to PoW in Nakamoto consensus, as long as all honest validators follow the protocol and own more than half of the total stake value, the probability of a block being revoked from the blockchain drops exponentially as the chain grows. From an economical perspective, attackers should be more reluctant to perform 51% attack in a PoS system than in a PoW system. In most PoS blockchain systems any fraudulent behavior of a validator is punishable by forfeiting its stake while in a PoW system only electricity is wasted. Therefore for most attackers losing all stakes is more economically devastating than wasting computing power.

Other security concerns Nonetheless, there are many other practical issues concerning the stability and security of chain-based PoS. Here we identify two of them.

1. **Time value of stake** PoS strongly resembles capitalism, where a dominant stakeholder can invest-profit-reinvest its capital and profit till a monopoly status. To alleviate the monopoly problem and encourage small stakeholder to participate in the game, a practical method is to let the unused stakes (of which the validators have not been a generator since the initial deposit) appreciate in value with time. Once a validator is chosen as the generator, its stake value returns to the default value at time zero. In the meantime the stakes of unchosen validators continue appreciating. In Peercoin for example, a validator's stake value is measured by *coin age*, which is the product of the deposited currency amount and the time elapsed since the initial stake deposit. The winning probabilities for small stakeholders grow in time as long as their stakes are kept unused. On the other hand, to prevent a stake from accumulating too much time value which can be exploited by a malicious validator to lock in a future block, the time value of a stake is limited by an upper bounded, for example 100 block generation cycles.

2. **Double-bet problem** This is also known as the *nothing-at-stake* problem. Since the longest-chain rule is still observed, when there are multiple parallel chain branches (forks), a PoS validator has the incentive to generate blocks on top of every branch at once without additional cost. In PoW, however, a miner has to do that by divesting its precious computing power to each additional branch. Therefore the chain-based PoS system need to incorporate a penalty scheme against those who place double bets. Possible choices include forfeiting the frozen stake, nullifying the block benefit for the correct bet, etc. However, these penalty schemes will be ineffective if a group of validators with more than 50% of the network's total stake value collude to maintain parallel chains.

2.4.2 BFT-based Consensus

BFT-based consensus protocols typically require high network connectivity and all nodes to reveal their true identities to others, a good fit for permissioned blockchains where the network size is small and the consortium of participants are known a priori. Similar to that of Nakamoto consensus, the goal of BFT-based consensus is to ensure all participants agree on a common history of blocks, which requires the correctness of block content and block order. However, there is a major difference between them: the *finality* condition for BFT-based consensus is deterministic. In other words, blocks will never been tampered once written into the blockchain.

PBFT for Blockchain

As we discussed in the previous section, PBFT is a classic consensus protocol for distributed computing based on state machine replication. To be used in a blockchain scenario, PBFT needs to adapt in the following ways.

1. **Parallelism** In PBFT replicas are separated into a primary and backups for every view. However the decentralization nature of blockchain requires that all nodes should be able to process client transactions as a primary and relay other's transactions. More specifically, when any node is ready to broadcast its new block, it initiates a new instance of PBFT by broadcasting a *Pre-Prepare* message containing this block. To deal with *Pre-Prepare* messages from different sources, the Prepare phase and Commit phase need to be modified in the way that the received blocks should be processed in chronological order. In other words, there can be multiple parallel protocol instances running and interacting in the Prepare and Commit phase.

2. **Dynamic view change** As there is only one primary in the original PBFT for each view, the view-change protocol can be executed in a rather orderly manner. In blockchain since every node can act as primary, the view-change protocol should be able to update multiple primaries in a single execution.

Theoretically, PBFT is able to tolerate f Byzantine nodes if the network size $N \geq 3f + 1$. In practical scenarios, there can be many implementation related issues preventing PBFT from realizing its full potential, with network connectivity being the major bottleneck. The operational messages in PBFT are time sensitive and a lowly connected network may not be able to execute PBFT in the correct manner. To make PBFT work most reliably, a fully connected network in required.

There are a handful blockchain initiatives using an adapted version of PBFT for consensus. Examples include Hyperledger Fabric[3] [15], Stellar [16]. Interested readers may refer their specific implementations of PBFT.

BFT-style PoS

BFT-style PoS has been used in Tendermint [17], EOS [18], and Ethereum's Casper initiative[4] [19]. Instead of following Nakomoto's contention-based blockchain generation process, BFT-style PoS embraces a more radical design in which the set of validators periodically finalize blocks in the main chain through BFT consensus. Here we use Ethereum Casper as an example. Similar to PBFT, Casper finalizes blocks from checkpoint to checkpoint. Each validator keeps a replica of the blockchain as well as a checkpoint tree. In every checkpoint cycle, Casper runs in following steps:

- **Step 1.** Every validator deposits an amount of currency (stake). The deposited stakes are kept frozen until the end of this checkpoint cycle.
- **Step 2.** Every validator grows new blocks from a justified checkpoint using a block proposal mechanism and then broadcasts them timely. No consensus is needed between validators at this time.
- **Step 3.** After a checkpoint interval is reached (100 blocks in Casper), the validators begin to form a consensus on a new checkpoint. Every validator casts a vote for a checkpoint block and broadcasts its vote to the network. The vote message contains five fields: hash of the source checkpoint s, hash of the voted target checkpoint t, height of s, height of t, and the validator's signature.
- **Step 4.** When a validator receive all votes, it reweighs the votes by sender's stake value and then computes the stake-weighted votes for each proposed checkpoint block. If a checkpoint t has a $2/3$ approval rate (super-majority), then the validator marks t *justified* the source checkpoint s *finalized*. All blocks before s are also finalized.

A fundamental difference between chain-based PoS and BFT-style PoS is that the latter offers deterministic finality. In other words, BFT-style PoS guarantees a finalized block will never be revoked in the future, while chain-based PoS and PoW don't rule out this possibility. Importantly, the deterministic

[3]Although PBFT is used currently, Hyperledger Fabric is designed to support an arbitrary consensus module in a plug-in fashion.

[4]The Ethereum Foundation plans to partially convert the Ethereum main net from PoW to Casper PoS by 2019.

finality also enables the punishment for double-betting validators (ie. solving nothing-at-stake problem). Because every finalized block comes with the proposer's public address, a validator is accountable for all finalized blocks it had proposed. Once it is found double-betting, the consensus protocol can legally forfeit the frozen stake of the double-betting validator and revoking the conflicting blocks.

Fault tolerance Since a proposed checkpoint needs a 2/3 approval rate to be justified, this algorithm can tolerate up to 1/3 faulty validators ideally. Nonetheless, due to the immaturity of PoS and specially the Casper PoS, there are many security and performance concerns that haven't been addressed. For example, what is the optimal checkpoint interval for the trade-off between security and communication efficiency, how to design a reliable and efficient block proposal mechanism without consensus until the next checkpoint, etc. The authors of this book will keep following the progress of Casper and other BFT-style PoS blockchains.

2.4.3 Proof-of-Elapsed-Time (PoET)

PoET concept was proposed by Intel in 2016 as an alternative to PoW. It is currently used in Hyperledger's Sawtooth project [20]. Compared to competing with computing power in PoW or currency ownership in PoS, PoET implemented a contention scheme based on a random back-off mechanism which has been widely used in medium access control protocols for local area networks. For a single block generation cycle, PoET is as simple as the following two steps:

- **Step 1.** Each validator waits a random length of time (back-off).
- **Step 2.** The first validator finishing the back-off becomes the generator.

Trusted random back-off To ensure the back-off time of each validator is truly random and fully elapsed, the back-off mechanism in each validator should be verified and trusted by all others. In practice, this can be achieved with a specially designed microprocessor that can execute sensitive programs in a trusted execution environment (TEE) or simply an "enclave". As as 2018, Intel and ARM are the market leaders for such microprocessors. Take Intel for example, some of its six-plus generation Core-series microprocessors are able to run Intel's Software Guard Extensions (SGX), which enables certain security services such as isolation and attestation [21]. In a PoET based blockchain, when a validator joins the network, it acquires the trusted back-off program from peers or a trusted server and runs it in a SGX-enabled enclave. If required by the trusted server, the validator can send its enclave measurement in an attestation report to the network indicating the authentic back-off program is loaded in its enclave. After successfully finishing a back-off, the validator proceeds to generate the new block; meanwhile the trusted back-off program in the enclave generates a certificate of completion along with the enclave measurement, which will be broadcast along with the new block.

Fault tolerance Theoretically, the PoET scheme can tolerate any number of faulty validators, as long as the back-off program running in a validator's enclave can be remotely attested by others, even if the hosting validator is not trustworthy. However, since each enclave runs the same back-off program independently, a rich validator can invest in multiple enclave instances to shorten its expected back-off time. This resembles PoW's economic model, with the only different that miners invest in TEE hardwares instead of mining devices. Therefore PoET needs to make sure more than 50% enclaves are in the hands of honest validators.

Hardware vendor dependency Another major drawback of PoET is its dependence on TEE platform providers, namely Intel and ARM, for providing TEE enabled hardwares and remote attestation services. Take Intel SGX for example, the security of the PoET system is bounded by the security of Intel's microprocessors and the reliability of Intel's attestation server. This explicit attack surface to some extent contradicts the blockchain's robustness-through-decentralization ideal.

2.4.4 Ripple

Operated by the Ripple company, Ripple is a real-time gross settlement network (RTGS) providing currency exchange and remittance services. Unlike public blockchain systems where anyone can participate in the validation process, Ripple regulates a set of known validators that mainly consist of companies and institutions. They run the Ripple server program and accept transaction requests from clients. A Ripple client only needs to submit transactions to their designated validator and the validator network will fulfill this transaction through consensus. Essentially, validators run the Ripple consensus protocol [22] in a distributed manner and form consensus on a common ledger of transactions.

Ripple consensus protocol

We will use "node" and "validator" interchangeably subsequently. In the validator network, each node p maintains a Unique Node List (UNL) of nodes, which is the only subnetwork p needs to trust partially (for not colluding). The Ripple consensus protocol is applied by each node for every consensus cycle. For each cycle, the protocol proceeds in four steps:

- **Step 1.** Each node prepares a *candidate set* containing all valid transactions it has seen, which may include new transactions submitted by clients and old transactions held over from the previous consensus cycle.
- **Step 2.** Each node combines its candidate set with the candidate sets of its UNL peers, votes "yes/no" on the validity of each transaction in the combined set, and sends votes to its UNL nodes.
- **Step 3.** Each node upon receiving votes from its UNL nodes, discards from its candidate set the transactions with a "yes" rate below a minimum threshold. The discarded transactions may be reused in the next consensus cycle.
- **Step 4.** Repeat step 2, 3 several rounds. In the final round, the threshold is increased to 80%. Each node appends the remaining transactions to its ledger and ends the consensus cycle.

Fault tolerance A transaction is finalized if it is approved by at least 80% nodes of the UNL. As long as $f \leq \frac{1}{5}(m-1)$ where m is the size of a UNL and f is the number of Byzantine nodes in the UNL, the Ripple consensus protocol is Byzantine-fault tolerant. This is a rather strong security assumption as it should be satisfied by every UNL-clique. In practice, this is fulfilled by Ripple's validator authentication scheme which ensures the true identity of any validator is known to others.

Connectivity requirement Since every node only keeps communication links to its UNL peers, different nodes may have disparate or even disjoint UNLs, which leads to the network partitioning problem as discussed previously. In a simple scenario, a group of nodes connected by UNL relationships can form a clique which is fully connected; however two UNL-cliques may agree on two conflicting ledgers in parallel if there is little communication between them. To prevent this problem, the Ripple network puts the following connectivity requirement for any two UNL cliques S_i, S_j:

$$|S_i \cap S_j| \geq \frac{1}{5} \max\{|S_i|, |S_j|\}, \quad \forall i, j \tag{2.2}$$

Table 2.6: A comparison of blockchain consensus algorithms

	PoW	**Chain-based PoS**	**BFT-style PoS**	**PoET**	**PBFT**	**Ripple Protocol**
Permission Needed	No	No	No	No	Yes	Yes
Third Party Needed	No	No	No	TEE platform vendor	Identity manager	Identity manager (Ripple Company)
Consensus Finality	Probabilistic	Probabilistic	Deterministic	Probabilistic	Deterministic	Deterministic
Connectivity Requirement	Low	Low	Low	Low	High	High
Fault Tolerance	50% hashing rate	50% stake value	33.3% stake value	50% enclave instances	33.3% voting power	20% nodes in UNL
Example	Bitcoin, Ethereum, Litecoin	Peercoin, Blackcoin	Ethereum Casper, Tendermint	Hyperledger Sawtooth	Hyperledger Fabric, Stellar	Ripple

This requirement literally means any pair of UNL cliques should share at least 25% of nodes. This level of inter-clique connectivity guarantees that no two UNL-cliques can agree on two conflicting transactions, because otherwise they would not pass the 80% approval requirement in the Ripple consensus protocol. Note that this connectivity requirement relies on Ripple company's supervision and thus is not realistic for public blockchains such as Bitcoin where there are more than ten thousand pseudonymous validators (miners).

Complexity analysis We assume every message has a fixed size, which is approximately the size of all transactions in the candidate set. Since a node only needs to communicate with its UNL peers, the message complexity of Ripple consensus protocol is $O(Km^2)$, where m is the size of UNL and K is the number of UNL cliques in the network.

2.5 Evaluation and Comparison

Table 2.6 qualitatively evaluates all the consensus protocols mentioned in this chapter. Specifically we consider the following aspects:

- **Permission needed:** *Yes* means the blockchain participants need to be authenticated at the joining and reveal true identities to others. *No* means any one join the network freely and pseudonymously.

- **Trusted third party needed:** Whether the network needs a trust third party for a common service.

- **Consensus finality:** The finality of blocks in the blockchain. *Probabilistic* means all written blocks (except the genesis block) are prone to revocation, although with small probabilities. *Deterministic* means all written blocks will never be revoked.

- **Connectivity requirement:** *Low* means a node only needs to maintain a minimum number of connections to peers. *High* means a node needs to connect with a significant percentage of the network.

- **Fault tolerance:** What percentage of faulty participants the protocol can tolerate. Different protocols have different adversarial models. For example hashing rate matters in PoW while stake value matters in PoS.

2.6 Summary

Consensus is a core function of a distributed system. We introduced the basics of distributed consensus, two practical consensus protocols for distributed computing, the basics of Nakamoto consensus, and several emerging blockchain consensus protocols. These consensus protocols are evaluated qualitatively and compared from security and complexity aspects. As of the year 2018, some of the protocols are still under development, such as Ethereum's Casper PoS, Hyperledger Sawtooth, and Hyperledger Fabric. And we will see more of them come out.

Generally, we need to take into account two models when designing a blockchain consensus protocol: network model and trust model. A highly connected and amenable network allows the participants to propagate transactions and blocks in a timely manner, which enables the use of message-heavy consensus protocols with high security guarantees. On the other hand, a benign trust model enables the utilization of highly efficient consensus protocols with focus on performance rather than security. The Nakamoto consensus protocol and PoW consensus algorithms in general have limit transaction capacity because they are deigned to endure uncertain network conditions and permissionless access scenarios with near-zero trust. In comparison, the BFT-based protocols and Ripple consensus protocol are highly efficient and support high transaction capacity because they are deigned for domain-specific applications in which high network connectivity is guaranteed and permissioned access is enforced.

In conclusion, consensus protocol is vital to the balance between security, performance, and efficiency for a blockchain system. A protocol designer needs to carefully consider the security requirement and performance target, as well as the level of communication complexity the network can undertake.

Acknowledgment

This work was supported in part by US National Science Foundation under grants CNS-1446478 and CNS-1443889.

Bibliography

[1] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics*, vol. 19. John Wiley & Sons, 2004.

[2] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[3] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[4] G. F. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design.* pearson education, 2005.

[5] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[6] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

[7] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proceedings of the 2nd international conference on Software engineering*, pp. 562–570, IEEE Computer Society Press, 1976.

[8] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[9] B. Liskov and J. Cowling, "Viewstamped replication revisited," 2012.

[10] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pp. 8–17, ACM, 1988.

[11] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[13] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pp. 1–10, IEEE, 2013.

[14] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," *self-published paper, August*, vol. 19, 2012.

[15] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," *arXiv preprint arXiv:1801.10228*, 2018.

[16] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus," *Stellar Development Foundation*, 2015.

[17] J. Kwon, "Tendermint: Consensus without mining," *Retrieved May*, vol. 18, p. 2017, 2014.

[18] "Eos.io technical white paper v2." `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`, 2018.

[19] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[20] "Hyperledger sawtooth project." `https://www.hyperledger.org/projects/sawtooth`, 2016.

[21] V. Costan and S. Devadas, "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[22] D. Schwartz, N. Youngs, A. Britto, *et al.*, "The ripple protocol consensus algorithm," *Ripple Labs Inc White Paper*, vol. 5, 2014.