SimpleFT: A Simple Byzantine Fault Tolerant Consensus

Rui Hao
Wuhan University of Technology
Wuhan, China

Chenglong Yi
Huazhong University of Science and Technology
Wuhan, China

Weiqi Dai
Huazhong University of Science and Technology
Wuhan, China

Zhaonan Zhang
Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Although having been popular for a long time, *Byzantine Fault Tolerance* (BFT) consensus under the partially-synchronous network is denounced to be inefficient or even infeasible in recent years, which calls for a more robust asynchronous consensus. On the other hand, almost all the existing asynchronous consensus are too complicated to understand and even suffer from the termination problem. Motivated by the above problems, we propose SimpleFT in this paper, which is a simple asynchronous consensus and is mainly inspired by the simplicity of the Bitcoin protocol. With a re-understanding of the Bitcoin protocol, we disassemble the life cycle of a block into three phases, namely proposal, dissemination, and confirmation. Corresponding to these phases, we devise or introduce the sortition algorithm, reliable broadcast algorithm, and quorum certificate mechanism in SimpleFT, respectively. To make full use of the network resources and improve the system throughput, we further introduce the layered architecture to SimpleFT, which enables multiple blocks to be confirmed at the same height. Comprehensive analysis is made to validate the correctness of SimpleFT and various experiments are conducted to demonstrate its efficient performance.

CCS CONCEPTS

• **Blockchain and Distributed Systems** → Consensus protocols;

KEYWORDS

consensus; Byzantine fault tolerance; blockchain; asynchronous network

1 INTRODUCTION

As the core component of blockchain systems, *Byzantine Fault Tolerance* (BFT) consensus protocols regain colossal interest during the past few years [36]. In general, the BFT consensus protocol (or consensus for short) enables a group of mutually distrustful replicas to reach an agreement [40], thus contributing to building replicated state machines [43]. According to different timing assumptions of the network, consensus protocols can be divided into three categories: synchronous consensus, asynchronous consensus, and partially-synchronous consensus. On the one hand, the assumption of a synchronous network can be easily violated, which may compromise the liveness or even safety properties [39]. On the other hand, as argued by the FLP theorem [19], it is impossible to design a deterministic consensus protocol in the asynchronous network. Therefore, for a long time, partial synchronization is picked as the timing assumption for most consensus designs.

The best-known representative of the partially-synchronous consensus protocols should be *Practical Byzantine Fault Tolerance* (PBFT) [30], which has been the de facto standard of consensus since its inception. Following PBFT, plenty of attempts are made to improve the protocol performance, such as AZyzzyva [22], Aardvark [16], and HotStuff [46]. However, as pointed out by Miller et al., partially-synchronous consensus protocols can be easily attacked to be infeasible or inefficient [35]. First, it is easy to construct an adversarial network scheduler to break the partially-synchronous assumption, thereby causing the system to a halt. Second, even if the partially-synchronous assumption is satisfied, it would take a long time to recover from a network partition, which leads to low performance.

To avoid the aforementioned problems of partially-synchronous consensus protocols, a line of recent works refocus on the asynchronous network assumption, such as HoneybadgerBFT [35], BEAT [17], and Dumbo [24]. To steer clear of the FLP impossibility theorem, these asynchronous consensus protocols usually introduce randomness to the protocol design. Concretely speaking, most of these asynchronous protocols integrate the *Asynchronous Byzantine Agreement* (ABA) algorithm [5], which is further based on the random common coin primitive [9]. In this regard, asynchronous protocols cannot guarantee the consensus to be reached in a deterministic round. Instead, they can only promise a consensus with a larger and larger probability.

However, the existing asynchronous consensus protocols are usually too complicated to understand, particularly compared with the simplicity of the Nakamoto consensus of Bitcoin [37]. An obscure protocol will also be difficult to implement, which usually brings more bugs and security problems [38]. Besides, the ABA algorithm widely adopted by existing asynchronous protocols suffers from the termination problem. To be more specific, replicas involved in an ABA algorithm cannot terminate in an elegant manner, some of which may run forever.

In this paper, we are also aimed to propose an asynchronous consensus protocol, named SimpleFT. Compared with the existing asynchronous protocols, SimpleFT is easier for understanding, without adopting the intricate and problematic ABA algorithm. On the whole, the design of SimpleFT is mainly inspired by the Bitcoin protocol, which inherits block and chain structures from Bitcoin. Since consensus protocols of SimpleFT and Bitcoin rely heavily on the blockchain structures, we henceforth use the terms 'consensus protocol' and 'blockchain protocol' interchangeably, which is also done by HotStuff [46] and Streamlet [13].

With a re-understanding of the Bitcoin protocol, we first disassemble the life cycle of a block into three phases, namely proposal, dissemination, and confirmation. Corresponding to these three phases, we then devise or introduce a multiple-round sortition algorithm, the *Reliable Broadcast* (RBC) algorithm, and the *Quorum Certificate* (QC) mechanism in SimpleFT. Concretely speaking, we devise a sortition algorithm to elect the block proposer at each height. Proposed blocks are disseminated through the RBC algorithm, which is then confirmed by the QC mechanism. The QC mechanism ensures that only one block will be confirmed at each height.

To avoid the waste of network resources and increase the system performance, we make an improvement to SimpleFT by enabling multiple blocks to be confirmed at each height. The improved SimpleFT is named layered SimpleFT, as a contrast to the basic SimpleFT described above. In layered SimpleFT, blocks are divided into two categories: *vote blocks* (VBlocks) and *data blocks* (DBlocks). VBlocks function to reach the consensus, while DBlocks only contain transactions. VBlocks are proposed after winning the multiple-round sortition algorithm and only one VBlock can be confirmed at each height, just like normal blocks in basic SimpleFT. By contrast, DBlocks are proposed without running a sortition algorithm, whose confirmations rely on the references by the confirmed VBlocks. In this way, multiple DBlocks can be referenced and confirmed at each height, thus improving the system performance.

We make a detailed analysis on the correctness of SimpleFT, including the safety and liveness properties. To evaluate SimpleFT's performance, we implement prototype systems of both the basic SimpleFT and layered SimpleFT. Various experiments are conducted based on prototype systems, whose results demonstrate SimpleFT's feasibility and efficiency. Besides, by comparing the performance of basic SimpleFT and layered SimpleFT, we can conclude that the layered architecture increases the system throughput by a substantial margin.

To sum up, this paper makes the following contributions:

- We carry out a disassembly of the Bitcoin protocol from the perspective of a block's life cycle, which includes three phases, namely proposal, dissemination, and confirmation.
- Inspired by the simplicity of Bitcoin, we propose a simple asynchronous consensus named SimpleFT.
- Corresponding to the three phases in Bitcoin, we devise or introduce the sortition algorithm, reliable broadcast algorithm, and quorum certificate mechanism in SimpleFT.
- Correctness analysis and various experiments are done to validate SimpleFT's correctness and efficiency, respectively.

2 PRELIMINARIES & MOTIVATION

On the whole, we aim to design a new consensus protocol in the asynchronous network. Before elaborating on our design, we first present the background and preliminary knowledge in this section. Particularly, we will discuss several timing assumptions to motivate our target scenario of an asynchronous network. After that, some algorithms integrated by SimpleFT are introduced.

2.1 Timing assumptions

In the context of distributed systems, the network can be divided into three categories according to different timing assumptions: synchronous network, asynchronous network, and partially-synchronous network [18]. Although the assumption of a synchronous network can greatly simplify the consensus design, this assumption can be violated with a large probability, especially in a world-scale *Wide Area Network* (WAN) deployment. Besides, the selection of the time-out parameter (Δ) for the synchronous network is a non-trivial task. If Δ is set too large, the consensus can be quite inefficient. On the contrary, a value of Δ that is too small will cause the synchronization assumption to be violated easily.

On the other hand, according to FLP impossibility theory, there will never be a deterministic asynchronous consensus protocol in the presence of one faulty replica [19]. As a result, for a long time, the assumption of a partially-synchronous network is widely adopted by researchers. The outstanding ones include PBFT [12], Zyzzyva [29], and HotStuff [46]. However, as discussed in [35] and [24], almost all the partially-synchronous consensus protocols are vulnerable to attacks of network partitions. To be more specific, the adversary can construct the network partitions when leaders are correct and repair the partition when leaders are Byzantine. In this way, the assumption of a partially-synchronous network is violated and no consensus can be reached.

Our protocol (i.e., SimpleFT) adopts the assumption of an asynchronous network, which promises better safety and liveness. Like many other asynchronous consensus protocols [24, 35], SimpleFT introduces randomness to the protocol design, thus circumventing the FLP impossibility theorem.

2.2 Reliable broadcast

In an asynchronous network under a Byzantine environment, broadcasting messages is not easy to do. In this regard, a Byzantine sender may send a message m to some replicas and a contradictory message m' to the others, which causes the correct replicas to deliver inconsistent messages. Besides, a sender may crash before sending the message m to all the replicas, where some correct replicas deliver m while others do not.

To describe a benign broadcast process in the asynchronous Byzantine network, an abstraction of *Reliable Broadcast* (RBC) is defined [7]. Particularly, RBC is characterized by the following three properties:

- *Validity*: If a correct replica broadcasts a message m, each correct replica will finally deliver m.
- *Consistency*: If two correct replicas deliver two messages m and m' respectively, then $m = m'$.
- *Totality*: If a message m is delivered by some correct replica, each correct replica will eventually deliver m.

Intuitively, with an RBC algorithm, all the correct replicas will eventually deliver the same messages. Besides, all the messages broadcast by a correct replica will eventually be delivered by each correct replica. The first practical RBC algorithm was proposed by Bracha [5], which was later improved by Cachin et al. [11] with the erasure code mechanism and Merkle tree structure [33]. Cachin's RBC algorithm opens a series of research on asynchronous consensus

protocols, which functions as an integral part of many remarkable works, such as HoneybadgerBFT [35] and Dumbo [24].

2.3 Verifiable random function

Verifiable Random Function (VRF) can be considered as a combination of random function and digital signature [34]. On one hand, VRF can output a random number given an input message. On the other hand, anyone can verify if the random number is created from a particular replica. To be more specific, taking a message (m) as the input, $\text{GenVRF}_{sk_i}(m)$ is expected to output a random value (x) and a proof (ψ), where sk_i denotes the private key of a particular replica (p_i). With the public key (pk_i) corresponding with sk_i, anyone receiving x can run the verification function $\text{VerifyVRF}_{pk_i}(m, \psi)$ to check if it is created from a VRF function by p_i.

VRF can be easily wrapped as a sortition function, which is further used to choose the committee members, as done by Algorand [21] and Blockene [42]. However, the existing sortition function can hardly guarantee to generate a committee of an exact size. For example, the VRF/sortition function in Blockene can only promise a committee of **approximate** 2000, which may be more or less. As for this paper, it is impossible for the VRF/sortition function to elect exactly one winner.

3 DISASSEMBLY OF BITCOIN

As a pioneer of blockchain systems, Bitcoin [37] reopens a boom of research on Byzantine consensus [30]. During the past decade, lots of wonderful works have been proposed with various network assumptions, such as SBFT [23] in the partially-synchronous network and HoneybadgerBFT [35] in the asynchronous network. Although these works promise better performance, they are becoming more and more complicated, making it difficult for nonspecialists to understand or implement. By contrast, **the Bitcoin protocol is concise enough, which can be easily grasped by laymen in a short time**. In this section, we disassemble the Bitcoin protocol into three phases according to the life cycle of a block, which inspires and instructs our protocol design.

3.1 Block proposal

In a blockchain system with no leaders, each replica is responsible for extending the chain. On the one hand, to guarantee the liveness property, there must be at least one replica to propose the new block at each height. On the other hand, to facilitate the safety property, it would be best if exactly one block is proposed at each height. Therefore, to resist the attacks of arbitrary block proposals at one height, there should be an eligibility checking for the proposals. To achieve these two objectives, Bitcoin makes a combination of the incentive mechanism and the *Proof of Work* (PoW) algorithm [25]. The incentive mechanism stimulates each replica to propose the new block by giving rewards to the eligible proposer, thus providing the liveness. A replica has to solve a PoW puzzle to prove its eligibility, which prevents arbitrary proposals. Besides, since it is scarcely possible for multiple replicas to solve the PoW puzzles simultaneously, the probability of multiple proposals for the same height is controlled at a low level.

INSIGHT 1. *There must be at least one block proposal for each height, preferably exactly one.*

3.2 Block dissemination

To enable each replica to possess the same data, a newly proposed block must be disseminated across the network. A proposer's willingness to broadcast a new block is also stimulated by the incentive mechanism, since the rewards become valid only if the new block is accepted by a majority of replicas. Another challenge is how to prohibit a proposer from sending inconsistent blocks to different replicas. Bitcoin deals with this by setting the PoW puzzle based on the block contents. In this way, any modifications to the block contents will create a totally different PoW puzzle. Therefore, it is impossible for a replica to broadcast two inconsistent blocks with only one pass of PoW computing. On the other hand, it would be uneconomical to solve the PoW puzzle multiple times at a single height. As for the data dissemination, Bitcoin adopts the Gossip protocol [26], which ensures a block to be delivered eventually by each replica.

INSIGHT 2. *The newly proposed blocks must be disseminated consistently and delivered eventually across the network.*

3.3 Block confirmation

As mentioned in Section 3.1, there may be multiple eligible block proposals at the same height. Further, these blocks of the same height may contradict each other, as they may contain conflicting transactions. In most blockchain systems (including Bitcoin), only one of these blocks is confirmed as valid. Therefore, a new question is how to confirm only one block for a height in a distributed manner securely. The safety property requires that if two blocks (B_k and B'_k) at the same height (k) are confirmed by two correct replicas, then $B_k = B'_k$. As for Bitcoin, it confirms the blocks based on the longest chain rule, where the blocks led by the longest chain are considered valid. As time goes by, the possibility that B_k and B'_k ($B_k \neq B'_k$) led by two chains of the same length is becoming smaller and smaller, thus guaranteeing the safety with overwhelming probability.

INSIGHT 3. *At each height, only one block can be confirmed consistently among all the correct replicas.*

4 BASIC SIMPLEFT

In this section, we first clarify the model of our consensus protocol. Following that, we elaborate on the basic version of SimpleFT, which is inspired by the disassembly of Bitcoin.

4.1 Model description

4.1.1 Replica setup. We consider a system consisting of $3f+1$ replicas, at most f of which are controlled by the malicious adversary, named as Byzantine replicas. Byzantine replicas can arbitrarily deviate from the protocol, while the others (correct replicas) will always conform with the protocol. We assume that the adversary is static and all the Byzantine replicas are determined by the adversary at the start of the protocol. Modern cryptography algorithms such as hash functions or signatures are employed. Therefore, all the replicas can be identified and certified by the *public-key infrastructure* (PKI). A message (m) signed by a replica (p_i) is denoted by $\langle m \rangle_i$. To reduce the message size of a collection of signatures, the threshold signature scheme is also adopted to merge the signatures from multiple parties [4].

4.1.2 Network assumption. The network is assumed to be asynchronous, where messages between two replicas are not guaranteed to be delivered within a predefined period. Each pair of replicas is connected via a *peer-to-peer* (P2P) link. Note that the adversary can only delay the messages transmitted between any pair of replicas but cannot remove them. Besides, the P2P link is pair-wised and authenticated that a message m received from the link connecting to replica p_i can be verified by the signature $\langle m \rangle_i$.

4.2 Protocol overview

Before describing the details, we give an overview of the basic SimpleFT protocol, including data structures and the life cycle of a block.

4.2.1 Data structures. Data structures in basic SimpleFT resemble Bitcoin to a great extent. To be more specific, transactions are packaged in the form of blocks. Each block is indexed with a height, which represents its distance from the genesis block. All the blocks are linked like a chain where a block B_k at height k includes the hash digest of its predecessor B_{k-1}. In terms of two blocks B_l and B_k, if B_l is a descendant of B_k, we say B_l extends B_k and denote it by $B_k \prec B_l$. Accordingly, $B_k \not\prec B_l$ represents that B_l does not extend B_k. If $B_k \neq B_j$ and $B_k \not\prec B_l$ and $B_l \not\prec B_k$, we say B_j is contradictory to B_k. Besides, we stipulate that the extension relation between blocks is reflexive, namely $B_k \prec B_k$.

Different from Bitcoin, SimpleFT requires a block proposer to contain an eligibility proof instead of a PoW nonce in the block. The eligibility proof is used to verify the qualification of the block proposer. Therefore, a block B_k proposed by an eligible replica p_i has the data structure as follows:

$$B_k := \langle b_k, H(B_{k-1}), \pi_k \rangle_i \tag{1}$$

where b_k denotes the plain block which only contains the outstanding transactions, H represents the hash function, and π_k denotes the eligibility proof generated for the height k.

4.2.2 Workflow of SimpleFT. SimpleFT operates in an epoch-by-epoch manner and epochs are divided according to the heights of blocks. In the remaining parts of this paper, we use the terms 'epoch' and 'height' interchangeably. Note that replicas in SimpleFT are not required to enter into or depart from the same epoch at the same pace, thus abiding by the assumption of an asynchronous network. As stated before, basic SimpleFT is designed by drawing an analogy to Bitcoin, where the epoch for each block is also divided into three phases. Concretely speaking, the **sortition algorithm** based on VRF is devised to propose the block, which is further disseminated via the **RBC** algorithm and confirmed by the **quorum certificate** mechanism. A comparison between Bitcoin and SimpleFT is shown in Table 1.

As an example, Figure 1 depicts three phases of a block in basic SimpleFT, where four replicas are involved. In the phase of block proposal, each replica will run the sortition algorithm locally. The sortition algorithm will return a boolean value, representing if the replica is eligible to propose a block. A replica who gets the eligibility is named as 'winner' to aid the presentation. Ideally, exactly one replica wins the sortition, as exemplified by p_1 in **Phase 1**. p_1 will then disseminate the new block to others via the RBC algorithm, as **Phase 2** in Figure 1 shows. After receiving the new block, each replica will vote for the new block. If $2f+1$ (i.e., 3 in this example) or more votes from distinct replicas are received, one can confirm the block. Besides, the $2f+1$ votes constitute a *Quorum Certificate* (QC), as shown by **Phase 3** in Figure 1.

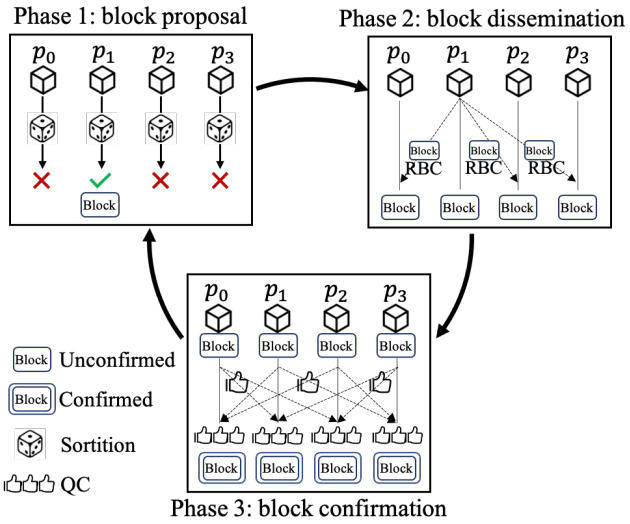


Figure 1: Three phases of a block in basic SimpleFT

Phases of a block	Bitcoin	SimpleFT
#1 Proposal	PoW & incentive	Sortition
#2 Dissemination	PoW & gossip	Reliable broadcast
#3 Confirmation	Longest chain	Quorum certificate

Table 1: Comparison between Bitcoin and SimpleFT

Algorithm 1 One-round sortition (for replica p_i)

Let k represent the epoch number and STR refer to the function converting an integer to a string.

function ONERUDSORT$_i(k)$
  $b, x, \psi \leftarrow$ SORTITION$_i$(STR(k))
  send $\langle b, x, \psi \rangle$ to BLKDISSEM$_i(k)$
  return
end function

function SORTITION$_i(m, e = 1)$
  $x, \psi \leftarrow$ GENVRF$_{sk_i}(m)$
  if $x \ mod \ (n/e) = 0$ **then**
    return *true*, x, ψ
  else
    return *false*
end function

4.3 Block proposal

A simple sortition algorithm based on VRF is described as Algorithm 1, where only one pass of the VRF function is executed. We

refer to this simple algorithm as one-round sortition, namely the ONERUDSORT function in Algorithm 1. We separate the core parts of the sortition algorithm as the SORTITION function, which will also be reused in Algorithm 2. The second argument e to the SORTITION$_i$ function is set as 1 by default, which denotes the easiness for a replica to win the SORTITION$_i$ function in a round. Besides, e also reflects the expected number of SORTITION$_i$'s winners when all the replicas are correct. However, Byzantine replicas may deviate from the protocol, such as refusing to run the SORTITION$_i$ algorithm, which reduces the expected number of winners in a round and increases the number of rounds needed to create a winner. This can further increase the consensus latency and reduce the system throughput. To remedy this problem, we enable the value of e to be adjusted dynamically, whose details will be presented in Section 7.1. Generally speaking, the larger the number of Byzantine replicas is, the larger the value of e will be. Besides, the dynamical adjustment of e is pretty similar to the adjustment of mining difficulty in Bitcoin. Note that if the SORTITION function outputs *true*, both the random value x and the proof ψ will be returned, seen as Line 10. By contrast, if the SORTITION function outputs *false*, only the boolean value needs to be returned, as Line 12 shows.

In the example taken by Figure 1, the sortition algorithm is assumed to produce exactly one winner, which is elected as the eligible block proposer. However, as discussed in Section 2.3, the simple one-round sortition algorithm cannot promise this. To be more specific, on the one hand, Algorithm 1 may produce no winner, thereby generating no blocks and compromising the liveness. On the other hand, it may produce more than one winner, thus generating multiple blocks for the same height/epoch. What's worse, due to the presence of Byzantine replicas, a Byzantine replica may deliberately refuse to propose a block, even if it wins the sortition. The above problems can be summarized as two challenges as follows:

CHALLENGE 1. *How to avoid the situation where no block is proposed, either because the sortition algorithm produces no winner or because a Byzantine winner refuses to propose a block?*

CHALLENGE 2. *How to deal with the situation where the sortition algorithm produces more than one winner?*

We propose a multiple-round sortition algorithm in Section 4.3.1 to deal with Challenge 1, while leaving Challenge 2 to be resolved in Section 4.4.

4.3.1 Multiple-round sortition. Given the SORTITION function is called with the default value of e, the probability that a replica wins the one-round sortition is $1/n$, where n=$3f$+1 represents the total number of replicas. The probability that at least one replica wins the one-round sortition is $1-(1-1/n)^n$. Moreover, the probability that at least one correct replica wins the one-round sortition is $1-(1-1/n)^{2f+1}$. Since a Byzantine replica may refuse to propose a block, the probability (P_{1+}) that at least one replica proposes the new block falls into an interval $[1-(1-1/n)^{2f+1}, 1-(1-1/n)^n]$. The lower bound of P_{1+} is $1-(1-1/n)^{2f+1}$, whose minimum is $1-e^{-2/3}$ where e is the mathematical constant.

Since one pass of the one-round sortition can generate a new block with a probability larger than $1-e^{-2/3}$, a natural thought is to keep running the one-round sortition over and over again, until a new block is generated. Following this thought, we devise a multiple-round sortition algorithm based on Algorithm 1, as shown by Algorithm 2. Roughly speaking, given a replica is aware that no valid block is proposed in the current round, it will strive to run next-round sortition and propose the block, as the **while** loop in Line 6 and $r \leftarrow r$+1 in Line 16 show. In this way, Challenge 1 can be addressed. However, a new issue is how to prevent the Byzantine replicas from running the sortition algorithm round and round infinitely, even if a valid block has been proposed for this epoch before.

Algorithm 2 Multiple-round sortition (for p_i)

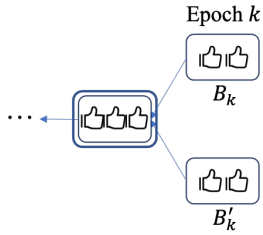
```
1:  Let CONCAT denote the function concatenating multiple strings
    and BROADCAST denote the general broadcast function. Let sig
    refer to a message channel connected to Line 4 in Algorithm 3.
2:  function MULRUDSORT_i(k)
3:      r ← 0; μ ← φ; V ← Φ; m ← STR(k)
4:      upon receiving s from sig of BLKDISSEM_i(k)
5:          return
6:      while true do
7:          upon receiving ⟨SORT, k, r⟩_j from p_j
8:              V ← V ∪ ⟨SORT, k, r⟩_j
9:          b, x, ψ ← SORTITION_i(m)
10:         if b then
11:             send ⟨b, x, ψ, r, μ⟩ to BLKDISSEM_i(k)
12:             return
13:         else
14:             BROADCAST(⟨SORT, k, r⟩_i)
15:             wait until |V| = 2f + 1
16:             r ← r+1; μ ← V; V ← Φ
17:             m ← CONCAT(STR(k), H(μ), STR(r))
18: end function
```

4.3.2 Prevention of infinite rounds by Byzantine. To prevent a Byzantine replica from running the sortition in infinite rounds, we require a replica to attach an extra proof named 'round proof' (μ), if it wants to run the r-th ($r \geq 1$) round sortition algorithm. The round proof for r-th round sortition is constructed based on the collected sortition results of $(r-1)$-th round.

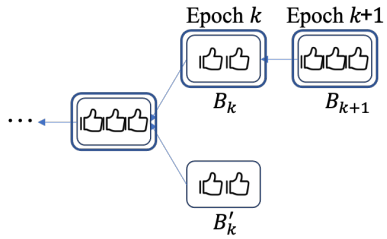
To be more specific, if the sortition result of the $(r-1)$-th round is *false*, each replica will broadcast the result, as shown by Line 14. To start the next round of sortition, a replica has to wait for messages from others. When there are $2f$+1 elements in V, V is taken as the round proof (μ) and the input for the next round of sortition is constructed based on the concatenation of k, μ, and r, as shown by Lines 15-17 in Algorithm 2. If the SORTITION function outputs *true*, not only b, x and ψ will be sent to the BLKDISSEM function, which is the same as Algorithm 1, but also the round number r and the round proof (μ) will be sent, as Lines 10-11 show. Besides, anytime when the replica receives a valid block for the epoch k from others, it will stop running the MULRUDSORT function, as shown by Lines 4-5.

4.4 Block dissemination & confirmation

As presented in Section 4.2, block dissemination is implemented based on RBC function, where the message type is set as BLK. The values (b, x, ψ, r, and μ) received from the MULRUDSORT function

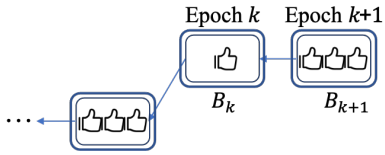


(a) Neither B_k nor $B_k{}'$ gets confirmed

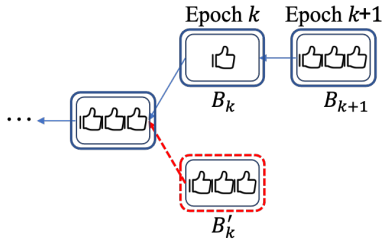


(b) B_k gets confirmed by B_{k+1}

Figure 2: Forks can be dealt with by descendant blocks



(a) B_k is extended and confirmed by B_{k+1}



(b) B'_k also gets confirmed

Figure 3: Multiple blocks in the same epoch are confirmed

constitute an eligibility proof (π) for the block proposal. Not only functioning to broadcast blocks from the replica (p_i) to others, the algorithm of the block dissemination also functions to receive blocks from others and send the signal to the channel *sig* in Algorithm 2. The pseudocode of the block dissemination is shown by Algorithm 3 in Appendix A.

A replica will vote for a block (B_k) if (1) it has not voted for the epoch k before and (2) B_k extends its local confirmed chain, whether the block is received from the RBC function or generated on its own. In other words, a replica will vote for one and only one block per epoch. The vote is also broadcast through the RBC function, with the message type as `VOTE`. During the RBC process for votes, correct replicas will multicast the `ECHO` messages only once for a sender in an epoch. In other words, it is impossible for a replica to broadcast two contradictory votes in an epoch. After receiving $2f+1$ votes for the same block, the block can be confirmed. However, since each replica runs the MULRUDSORT function in parallel, there may be multiple valid blocks generated in the same epoch, as Challenge 2 presents. Each of these same-epoch blocks leads a chain, resulting in forks just like Bitcoin. Furthermore, since there are at most $3f+1$ votes in an epoch, maybe none of these blocks can get $2f+1$ or more votes, in which situation no block can be confirmed directly.

To deal with this challenge, we allow a replica to start the process of the next epoch and strive to propose a new block to extend the longest chain, even if the last block has not yet been confirmed. If there are multiple longest chains with the same length, the replica can randomly pick one or simply pick the first received. As long as the new block can get $2f+1$ votes, both this block and all its ancestors can be confirmed. An example to demonstrate the challenge of chain forks and their solution is shown in Figure 2. Assume there are four replicas and two valid blocks (B_k and B'_k) are generated in epoch k. However, each block can only get $2f$ (i.e., 2) votes and cannot get confirmed directly, as Figure 2a shows. In this regard, a replica having received B_k or B'_k can attempt to propose the block in epoch $k+1$. If a block B_{k+1} is proposed to extend B_k and gets at least $2f+1$ votes, it and its predecessors can be confirmed, as shown by Figure 2b. The mechanism that all the ancestors can be confirmed by a descendant block is named as *ancestor confirmation* mechanism, which deals with Challenge 2 successfully.

However, the aforementioned solution to Challenge 2 poses a new challenge. An example is shown by Figure 3, where a replica p_i proposes a new block B_{k+1} after receiving the block B_k. Although B_k cannot be confirmed by $2f+1$ direct votes, it is confirmed by the descendant B_{k+1} indirectly, as Figure 3a shows. On the other hand, a contradictory block B'_k can also get $2f+1$ votes to be confirmed. In this manner, multiple contradictory blocks in the same epoch are confirmed, as shown by Figure 3b, which compromises the safety. This challenge can be summarized as follows:

CHALLENGE 3. *How to avoid multiple contradictory blocks in the same epoch from being confirmed, after taking the ancestor confirmation mechanism?*

To tackle Challenge 3, we require a replica to propose the block of the next epoch ($k+1$) only after it has collected $2f+1$ (not necessarily consistent) votes for epoch k, which constitute a set $S(k)$. Based on $S(k)$, the replica chooses which precursor block to extend according to the rules described in Equation 2. Once the precursor block is chosen, the replica can start the block proposal for epoch $k+1$ immediately.

$$Pre(k) = \begin{cases} B_k, & |L(k)| \geq f+1 \;\&\&\; L(k) \text{ votes for } B_k \\ \texttt{RAN}, & \text{otherwise} \end{cases} \tag{2}$$

where $L(k)$ represents the largest subset of $S(k)$ voting for the same block and `RAN` denotes a randomly chosen block. To be more specific, if a block in the epoch k gets $f+1$ or more votes, it will be chosen as the precursor block to extend. Otherwise, the precursor block is chosen at random. In fact, once a replica receives $f+1$ consistent votes for a block, the replica can choose this block immediately without collecting $2f+1$ votes, thus reducing the waiting

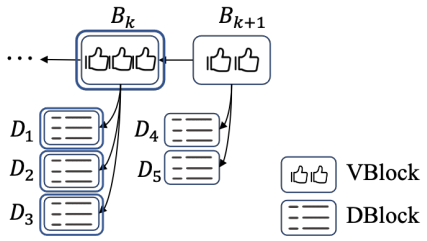


Figure 4: The architecture of layered SimpleFT

latency. To prevent the Byzantine replicas from doing evil, the block proposal must be attached with a proof for the precursor choice, namely precursor proof ξ, as shown by Line 7 in Algorithm 3 in Appendix A. In fact, the combination of the sortition proof ψ, the proof μ, and the precursor proof ξ makes up the complete eligibility proof π in Equation 1.

5 LAYERED SIMPLEFT

In the basic SimpleFT protocol described in Section 4, only one block in each epoch will be confirmed. In other words, all the blocks except one will be discarded, which causes a waste of network resources since each block has already been disseminated through the expensive RBC function. To address this problem, we introduce the layered architecture to SimpleFT, which reduces the network resource wastage and improves the system throughput largely. Blocks in layered SimpleFT are divided into two categories, namely *data block* (DBlock) and *vote block* (VBlock), where DBlocks are referenced by VBlocks. If a VBlock is confirmed by $2f+1$ votes, all its referenced DBlocks are confirmed. Therefore, there can be only one confirmed VBlock but multiple confirmed DBlocks in each epoch. As shown by Figure 4, DBlocks (D_1, D_2, and D_3) are confirmed as a result of the confirmed VBlock (B_k). By contrast, either D_4 or D_5 has not yet been confirmed.

5.1 Data blocks

A *data block* (DBlock) contains only the plain transactions, which is also broadcast through the RBC algorithm. Each replica can package a DBlock based on the outstanding transactions without running the sortition algorithms. To avoid a Byzantine replica from packaging the DBlocks arbitrarily, the DBlock is required to be attached with a sequence number, which equals the epoch number of its latest received VBlock. Therefore, a DBlock (D) packaged by replica p_i is in the format of $\langle txs, sn\rangle_i$, where txs and sn represent the packaged transactions and sequence number respectively. Accordingly, the RBC algorithm is modified by adding a validation mechanism, as shown by Algorithm 6 in Appendix A. Concretely speaking, a replica p_j will respond to the `VAL` message for DBlock $\langle txs, sn\rangle_i$ if (1) it has not processed a `VAL` message for sn from p_i before and (2) $sn \leq e + w$, where e represents the epoch number of p_j's latest received VBlock. Besides, w denotes an empirical value to control how fast DBlocks are proposed, which can be simply set as 10. Each replica will maintain a pool to store DBlocks that have been received from the RBC algorithm but not yet confirmed by VBlocks.

5.2 Vote blocks

A *vote block* (VBlock) is quite similar to the normal block in basic SimpleFT, except that it contains references to the DBlocks rather than the outstanding transactions. Particularly, if a replica gets the eligibility from the sortition algorithm, it will propose a VBlock referencing all the DBlocks in its DBlock pool. However, a Byzantine replica may maliciously reference a DBlock (e.g., D_b), which is not yet broadcast through the RBC algorithm, leading other correct replicas to be in a dilemma. On the one hand, the correct replica should stop running the MULRUDSORT algorithm for the current epoch due to the receipt of a VBlock. On the other hand, it may not be able to receive and validate the DBlock D_b referenced by this VBlock, since D_b is not broadcast through the RBC algorithm at all. In this regard, the correct replica can never vote for this VBlock and the system can be trapped in a halt.

To prevent the Byzantine replica from referencing a non-RBC DBlock, we require the VBlock to contain proof for each referenced block. Particularly, we make use of the Provable Reliable Broadcast (PRBC) algorithm [24] to broadcast the DBlocks, which outputs a threshold signature σ by combining share signatures from $f+1$ replicas. The share signature from a replica indicates that the replica has received the DBlock through the RBC algorithm. Since there are at most f Byzantine replicas, σ can be used as proof that all the correct replicas will eventually receive the DBlock. Therefore, the VBlock has a data structure as follows:

$$B_k := \langle H(B_{k-1}), \pi_k, \{\langle H(D_0), \sigma_0\rangle, ..., \langle H(D_l), \sigma_l\rangle\}\rangle_i$$

Each confirmed DBlock D referenced by a VBlock B can be identified by a tuple $\langle k, d\rangle$, namely $D_{\langle k,d\rangle}$, where k and d represent the epoch number of B and the serial number of D in B. After receiving B_k, each replica will vote for it only if both the eligibility proof π_k and the PRBC proofs σ pass the validation.

5.3 Basic SimpleFT vs. layered SimpleFT

In fact, basic SimpleFT can be considered a special case of layered SimpleFT. To be more specific, basic SimpleFT introduces several simplifications or modifications to layered SimpleFT as follows:

- Each VBlock can only reference one DBlock.
- The DBlock and the VBlock are broadcast simultaneously, through the same RBC algorithm.
- Neither the PRBC algorithm nor the PRBC proof σ in the VBlock is needed.

Since basic SimpleFT is a special case of layered SimpleFT, we carry out a generalization to the notions described in Section 4.2. The extension relationship between VBlocks is defined similarly to that between general blocks in basic SimpleFT. Namely, we say the VBlock B_l extends B_k ($B_k \prec B_l$), if B_l is a descendant of B_k. As for two DBlocks $D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$, we say $D_{\langle l,e\rangle}$ extends $D_{\langle k,d\rangle}$ (i.e., $D_{\langle k,d\rangle} \prec D_{\langle l,e\rangle}$) if the following condition is satisfied:

$$B_k \prec B_l \;||\; (B_k = B_l \;\&\&\; d < e) \tag{3}$$

It is easy to find that all the confirmed DBlocks can be sorted by the extension relationship ($\prec$) in total order.

5.4 Transaction execution

Since DBlocks are created in parallel, there may be duplicate or even conflicting transactions being packaged in different DBlocks, which are named conflicting blocks. In layered SimpleFT, we allow the VBlock to reference these conflicting blocks. To resolve the conflicting transactions, we further generalize the extension relationship from DBlocks to transactions. Each confirmed transaction T contained in the DBlock $D_{\langle k,d\rangle}$ can be identified by a tuple $\langle k, d, u\rangle$, denoted by $T_{\langle k,d,u\rangle}$, where u represents the sequence number of T in $D_{\langle k,d\rangle}$. As for two transactions $T_{\langle k,d,u\rangle}$ and $T_{\langle l,e,v\rangle}$, we say $T_{\langle l,e,v\rangle}$ extends $T_{\langle k,d,u\rangle}$ (i.e., $T_{\langle k,d,u\rangle} \prec T_{\langle l,e,v\rangle}$), if the following condition is satisfied:

$$D_{\langle k,d\rangle} \prec D_{\langle l,e\rangle} \;||\; (D_{\langle k,d\rangle} = D_{\langle l,e\rangle} \;\&\&\; u < v) \tag{4}$$

It is also easy to conclude that all the confirmed transactions can be sorted by the extension relationship ($\prec$) in total order. With this order, each confirmed transaction can be attached with an index x and this transaction can be named as being confirmed at x. If $T_{\langle k,d,u\rangle}$ and $T_{\langle l,e,v\rangle}$ are confirmed at indexes x and y respectively, and $T_{\langle k,d,u\rangle} \prec T_{\langle l,e,v\rangle}$, we must have $x < y$. Only the first one of the conflicting transactions will be executed. Take two conflicting transactions $T_{\langle k,d,u\rangle}$ and $T_{\langle l,e,v\rangle}$ as an example. If $T_{\langle k,d,u\rangle} \prec T_{\langle l,e,v\rangle}$, $T_{\langle k,d,u\rangle}$ will be considered as valid and executed, while $T_{\langle l,e,v\rangle}$ will be discarded without execution. In other words, the confirmed DBlocks in layered SimpleFT will contain invalid transactions, which is similar to Ethereum [45] and Hyperledger Fabric [2].

6 CORRECTNESS ANALYSIS

Since basic SimpleFT can be considered as a special case of layered SimpleFT, we analyze the correctness of layered SimpleFT directly. The correctness of a consensus protocol includes two aspects, namely safety and liveness. To aid presentation, we say a VBlock B is confirmed directly, if it is confirmed by $2f+1$ votes for B. On the contrary, we say B is confirmed indirectly if it is confirmed by its descendant. A VBlock being confirmed directly is also named as QCVBlock.

6.1 Safety

LEMMA 1. *If two VBlocks (B_k and B_l) are confirmed, then either $B_k = B_l$, $B_k \prec B_l$, or $B_l \prec B_k$.*

PROOF. Let C_x denote the lowest QCVBlock to confirm B_k, where x represents the QCVBlock's epoch number. That means, if B_k is confirmed directly, $C_x = B_k$; if not, C_x is the lowest QCVBlock extending B_k. Similarly, let C_y denote the lowest QCVBlock to confirm B_l. Apparently, $B_k \prec C_x$ and $B_l \prec C_y$. If $x = y$, it means both C_x and C_y are directly confirmed, namely get $2f+1$ votes, in the same epoch. In this situation, C_x must equal C_y ($C_x = C_y$). Otherwise, there must be at least one correct replica voting for two contradictory VBlocks in the same epoch, which is impossible.

Without loss of generality, we assume $x < y$. Next, we prove that if $x < y$, we must have $C_x \prec C_y$. Suppose for contradiction that $C_x \not\prec C_y$. There must be an ancestor block of C_y in the epoch x, denoted by C'_x. We have $C'_x \prec C_y$ and $C'_x \neq C_x$. According to the precursor choice rules defined in Equation 2, when C'_x was chosen as the precursor block to extend, either C'_x gets at least $f+1$ votes (case 1) or none of the blocks gets $f+1$ or more votes (case 2). As for case 1, since C_x is directly confirmed by $2f+1$ votes, there must be one replica vote for two contradictory blocks (i.e., C_x and C'_x) in the same epoch through the RBC function. However, according to the definition of RBC function, a replica can only vote for one block in an epoch. Therefore, case 1 is impossible. In terms of case 2, none of the blocks gets $f+1$ or more votes in the collected $2f+1$ votes. Taking the remaining f votes into consideration, none of the blocks can finally get $2f+1$ or more votes. Therefore, it is impossible for C_x to get confirmed directly. To sum up, the assumption that $C_x \not\prec C_y$ is invalid, thus $C_x \prec C_y$.

On the one hand, when either $C_x = C_y$ or $C_x \prec C_y$, we have $B_k \prec C_x \prec C_y$, namely B_k is an ancestor of C_y. On the other side, since $B_l \prec C_y$, B_l is also an ancestor of C_y. Combining two sides together, we must have either $B_k = B_l$, $B_k \prec B_l$, or $B_l \prec B_k$.

LEMMA 2. *If two DBlocks ($D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$) are confirmed, then either $D_{\langle k,d\rangle} = D_{\langle l,e\rangle}$, $D_{\langle k,d\rangle} \prec D_{\langle l,e\rangle}$, or $D_{\langle l,e\rangle} \prec D_{\langle k,d\rangle}$.*

PROOF. Denote the VBlocks corresponding to these two DBlocks as B_k and B_l respectively. Since $D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$ are confirmed, both B_k and B_l must be confirmed as well. According to Lemma 1, either $B_k = B_l$, $B_k \prec B_l$, or $B_l \prec B_k$. On the one hand, according to Equation 3, if $B_k \prec B_l$ or $B_l \prec B_k$, we have $D_{\langle k,d\rangle} \prec D_{\langle l,e\rangle}$ or $D_{\langle l,e\rangle} \prec D_{\langle k,d\rangle}$. On the other hand, if $B_k = B_l$, it means $D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$ are referenced by the same VBlock. If $d = e$, $D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$) refer to the same DBlock and thus $D_{\langle k,d\rangle} = D_{\langle l,e\rangle}$. Otherwise, $D_{\langle k,d\rangle} \prec D_{\langle l,e\rangle}$ or $D_{\langle l,e\rangle} \prec D_{\langle k,d\rangle}$, according to Equation 3. To sum up, if $D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$ are confirmed, then either $D_{\langle k,d\rangle} = D_{\langle l,e\rangle}$, $D_{\langle k,d\rangle} \prec D_{\langle l,e\rangle}$, or $D_{\langle l,e\rangle} \prec D_{\langle k,d\rangle}$.

LEMMA 3. *If two transactions $T_{\langle k,d,u\rangle}$ and $T_{\langle l,e,v\rangle}$ are confirmed, then either $T_{\langle k,d,u\rangle} = T_{\langle l,e,v\rangle}$, $T_{\langle k,d,u\rangle} \prec T_{\langle l,e,v\rangle}$, or $T_{\langle l,e,v\rangle} \prec T_{\langle k,d,u\rangle}$.*

PROOF. Denote the DBlocks corresponding to these two transactions as $D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$ respectively. Since $T_{\langle k,d,u\rangle}$ and $T_{\langle l,e,v\rangle}$ are confirmed, both $D_{\langle k,d\rangle}$ and $D_{\langle l,e\rangle}$ must be confirmed as well. According to Lemma 2, either $D_{\langle k,d\rangle} = D_{\langle l,e\rangle}$, $D_{\langle k,d\rangle} \prec D_{\langle l,e\rangle}$, or $D_{\langle l,e\rangle} \prec D_{\langle k,d\rangle}$. Similar to the proof for Lemm2, according to Equation 4, we have either $T_{\langle k,d,u\rangle} = T_{\langle l,e,v\rangle}$, $T_{\langle k,d,u\rangle} \prec T_{\langle l,e,v\rangle}$, or $T_{\langle l,e,v\rangle} \prec T_{\langle k,d,u\rangle}$.

THEOREM 4 (SAFETY). *If two transactions T and T' are confirmed at the same index, then $T = T'$.*

PROOF. Since T and T' are confirmed, according to Lemma 3, we have either $T = T'$, $T \prec T'$, or $T' \prec T$. Suppose for contradiction that T and T' are confirmed at the indexes x and y, respectively. If either $T \prec T'$ or $T' \prec T$, we have $x < y$ or $y < x$, which is contradictory to the condition that T and T' are confirmed at the same index. Therefore, T must equal T'.

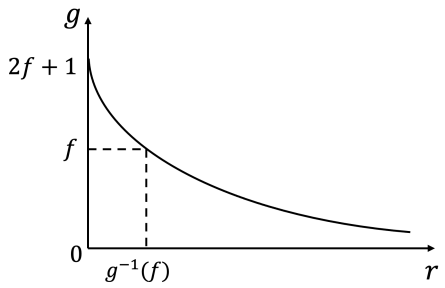


Figure 5: Schematic diagram of g and $g^{-1}(f)$

6.2 Liveness

Generally speaking, the liveness property of the blockchain system requires that a transaction can be eventually confirmed. In the context of SimpleFT, if the VBlocks can be confirmed after a while, new transactions can be continuously confirmed. Furthermore, since a VBlock is confirmed by itself being QCVBlock directly or by its descendant QCVBlock indirectly, we transform the liveness property as Theorem 6.

LEMMA 5. *Denote P as the probability that exactly one VBlock is disseminated in an epoch. We have that P is not a infinitely small value, namely $P \not\to 0$.*

PROOF. Let p be the probability that the one-round sortition algorithm (i.e., SORTITION in Algorithm 1) returns *true*. Let P_0 be the probability that exactly one correct replica wins in the 0-th round (round 0) of the MULRUDSORT algorithm, which can be calculated as follows:

$$P_0 = \sum_{i=1}^{2f+1} p(1-p)^{3f} = (2f+1)\cdot p(1-p)^{3f} \tag{5}$$

Let P_0^r represent the probability that exactly one correct replica wins in round 0 while all the other replicas fail in the rounds from 1 to r of MULRUDSORT. We remark that a replica with no round proof to run the sortition of round r^o ($1 \le r^o \le r$) is considered to fail in round r^o with a probability of 1. Besides, we stipulate that $P_0^0 = P_0$.

Let $g(r)$ be the number of correct nodes having not returned from MULRUDSORT at the beginning of round r. Consider the situation where exactly one correct replica wins in round 0 of MULRUDSORT, $g(r)$ should decrease as r increases. In other words, $g(r)$ is a concave decreasing function of r. Besides, $g(0) = 2f+1$ and $g(\infty) = 0$. Let $g^{-1}(v)$ be the inverse function of $g(r)$ and $g^{-1}(f)$ represents the minimum round r whose $g(r)$ is no more than f. Since $g(r)$ is a decreasing function, for each r that $r \ge g^{-1}(f)$, we must have $g(r) \le f$. An example to demonstrate the function $g(r)$ and the value $g^{-1}(f)$ is shown by Figure 5.

Recall that a replica is qualified to start the next round (r+1) of MULRUDSORT only if it collects $2f$+1 `SORT` messages for r. Once $g(r) \le f$, the number of correct replicas to broadcast `SORT` messages in round r will be at most f. Therefore, neither a correct replica nor a Byzantine replica can start the round $r+1$ of MULRUDSORT. In other words, for each r satisfying the condition $g(r) \le f$, we have $P_0^{r+1} = P_0^r$. On the other hand, for each r satisfying the condition $g(r) \ge f$+1, we have $P_0^{r+1} \ge P_0^r \cdot (1-p)^{g(r)+f}$. Namely, P_0^r can be figured out as follows:

$$P_0^{r+1} \ge P_0 \cdot \prod_{u=0}^{min(r,g^{-1}(f))} \left[(1-p)^{g(u)+f}\right], \forall r \ge 0 \tag{6}$$

Therefore, P can be calculated according to Equation 7.

$$\begin{aligned} P &\ge \lim_{r\to\infty} P_0^{r+1} \\ &\ge P_0 \cdot \prod_{u=0}^{g^{-1}(f)} \left[(1-p)^{g(u)+f}\right] \\ &\ge P_0 \cdot \prod_{u=0}^{g^{-1}(f)} \left[(1-p)^{2f+1+f}\right] \\ &= P_0 \cdot \left[(1-p)^{3f+1}\right]^{g^{-1}(f)+1} \\ &= (2f+1)\cdot p(1-p)^{3f} \cdot \left[(1-p)^{(3f+1)\cdot(g^{-1}(f)+1)}\right] \end{aligned} \tag{7}$$

Since p is not infinitely small (i.e., $p \not\to 0$) and $g^{-1}(f)$ is not infinitely large (i.e., $g^{-1}(f) \not\to \infty$), P will never be infinitely small, namely $P \not\to 0$.

THEOREM 6 (LIVENESS). *Let $\lambda(t)$ be the probability that there will be at least one QCVBlock during the period t after any time point. We have* $\lim_{t\to\infty} \lambda(t) \to 1$.

PROOF. Without loss of generality, we represent the 'any time point' by the time when a QCVBlock (e.g., with epoch number as e) is constituted and denote the period t by the number (m) of epochs after e. To prove Theorem 6, we only have to prove that $\lim_{m\to\infty} \gamma(m) \to 1$, where $\gamma(m)$ represents the probability that there will be at least one QCVBlock in the following m epochs after e.

If there is only one VBlock (B_k) is disseminated in epoch k, B_k will definitely receive $2f$+1 votes, thus turning into a QCVBlock. Therefore, for an epoch k that $k > e$, the probability (ζ_k) that a VBlock B_k turns into a QCVBlock is at least P (i.e., $\zeta_k \ge P$), where P is defined in Lemma 5. Thus, $\gamma(m)$ can be calculated according to Equation 8.

$$\begin{aligned} \gamma(m) &= 1 - \prod_{k=e+1}^{e+m} (1-\zeta_k) \\ &\ge 1 - \prod_{k=e+1}^{e+m} (1-P) \\ &\ge 1 - (1-P)^m \end{aligned} \tag{8}$$

Namely, we have:

$$\begin{aligned} \lim_{m\to\infty} \gamma(m) &\ge \lim_{m\to\infty} \left(1-(1-P)^m\right) \\ &= 1 - \lim_{m\to\infty} (1-P)^m \end{aligned} \tag{9}$$

According to Lemma 5 that $P \not\to 0$, we have $1 - P \not\to 1$ and $\lim_{m\to\infty}(1-P)^m \to 0$. Therefore, $1 - \lim_{m\to\infty}(1-P)^m \to 1$ and $\lim_{m\to\infty} \gamma(m) \to 1$. Equivalently, $\lim_{t\to\infty} \lambda(t) \to 1$ and thus Theorem 6 is proved.

7 IMPLEMENTATION AND EVALUATION

In this section, we talk about the experiments to evaluate both the basic SimpleFT and layered SimpleFT. First, we detail the prototype implementation and the experimental settings. Based on the prototypes, we evaluate the performance of our design, whose metrics mainly include throughput and latency.

7.1 Implementation

Prototypes of both the basic SimpleFT and layered SimpleFT are implemented in Golang[1], with a total of around 3,600 lines of code. We implement the MULRUDSORT function based on Yoseplee's open-source VRF library[2]. The RBC algorithm is implemented according to the improved version proposed by Bracha [5], which absorbs the Merkle tree structure and erasure coding mechanism as ingredients. Pseudocodes of RBC for VBlocks and votes are shown as Algorithm 4 and Algorithm 5 in Appendix A, respectively. The parts of erasure coding are implemented based on Reed-Solomon library[3]. During the process of implementation, we also refer to the project of HoneybadgerBFT[4]. As for the PRBC algorithm to disseminate DBlocks, we adapt the algorithms of Dumbo [24], whose pseudocodes are shown in Algorithm 6. To reduce the message complexity, we introduce the threshold signature to combine signatures from multiple replicas into one. Particularly, we employ the DEDIS advanced crypto library[5] to implement the threshold signatures.

Recall that in Section 4.3, the second argument e to the SORTITION function is adjusted dynamically to deal with the arbitrary actions taken by Byzantine replicas and maintain the system throughput. In the prototype implementation, we simply adjust the value of e every 1,000 VBlocks. To be more specific, we calculate the average ($\bar{r}$) of the rounds needed to win the MULRUDSORT algorithm in the last 1,000 VBlocks. The value of e for the next 1,000 epochs can be calculated as follows:

$$e = min(e' \cdot \bar{r}, \quad \frac{3}{2}) \tag{10}$$

where e' represents the argument in the last 1000 epochs. The reason why e should be less than $3/2$ is that there are at least $\lceil (2n)/3 \rceil$ correct replicas. Therefore, to make the expected number of winners be 1, the probability for a replica to win the SORTITION algorithm should be roughly less than $3/(2n)$ and e should be less than $3/2$.

7.2 Experimental setting

We deploy the prototype systems on the Alibaba cloud. Each replica is run on an ECS.c6e.2xlarge instance, which contains 8 vCPU and 16 GB memory. To characterize the actual deployment environment, replicas are uniformly distributed in five regions around the world. All the replicas are connected via the link of 100 Mbps bandwidth.

Since we are the first to design the asynchronous consensus by imitating Bitcoin, there is no perfect counterpart for comparison. Worse still, we can hardly find the publicly available implementations of existing consensus works. As PBFT has been considered the de facto standard of the consensus protocol for a long time, which is proved to be efficient enough, we mainly take PBFT as our comparison counterpart. Besides, although there is an open-source implementation of PBFT (i.e., BFT-SMaRt[6]), it is reported to be poorly scalable [44], which can only work in a WAN environment consisting of no more than four replicas [23]. Therefore, we implement a well-scalable version of PBFT by ourselves, which also absorbs the threshold signature to optimize the message complexity, as presented in SBFT [23].

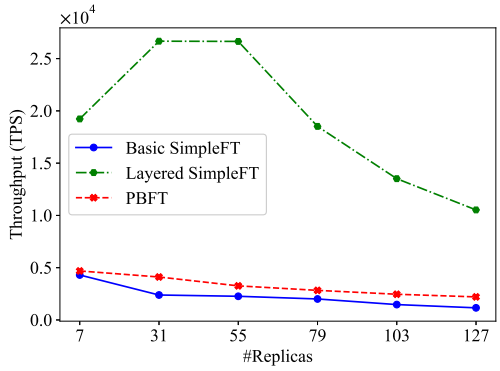


Figure 6: Throughput comparison between PBFT, basic SimpleFT, and layered SimpleFT

7.3 Throughput comparison

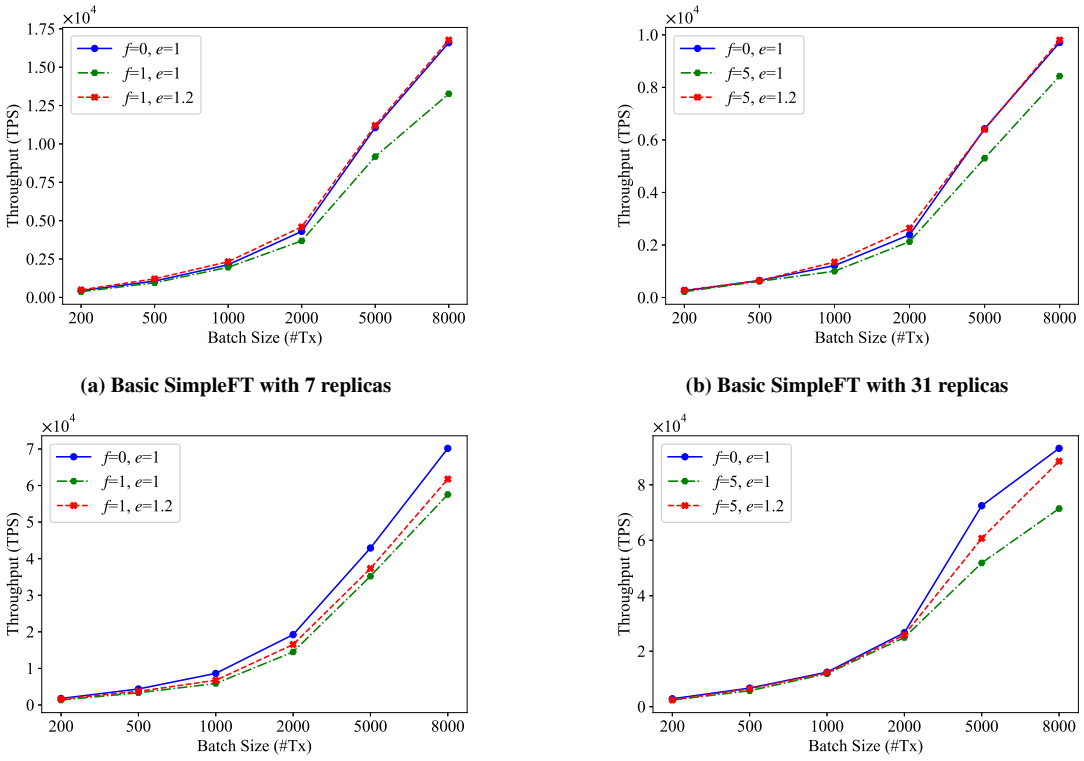
To get an intuitive understanding of SimpleFT's performance, we first compare the throughput of PBFT, basic SimpleFT, and layered SimpleFT. In this section, we simply set the batch size, namely the number of transactions in a batch, as 2,000. Evaluation of the throughput changes resulting from the increase in batch sizes remains to be made in the next section. Besides, all the replicas are set as correct ones.

Experimental results are shown in Figure 6, with the number of replicas as x-axis. In general, the basic SimpleFT is a little inferior to PBFT. The reason is that SimpleFT needs a sortition algorithm before starting a consensus, which introduces higher latency and reduces the throughput. However, compared with the benefits brought by the asynchronous consensus, the performance penalty of basic SimpleFT is acceptable, which obtains 71.7% throughput of PBFT. On the other hand, layered SimpleFT substantially outperforms both PBFT and basic SimpleFT. Particularly, when the system consists of 55 replicas, layered SimpleFT delivers 8.2x and 11.8x throughput over PBFT and basic SimpleFT, respectively. An abnormal phenomenon in Figure 6 is that the throughput of layered SimpleFT experiences a rise and then a fall as the number of replicas increases. The throughput rise results from more DBlocks being created and confirmed in one epoch, while the fall is due to the limitation of the network bandwidth. Too many DBlocks being propagated simultaneously will cause network congestion, thus slowing down the consensus process.

7.4 Robustness of throughput

As described in Section 7.1, to remedy the throughput reduction problem resulting from the Byzantine replicas, we introduce an

[1] https://go.dev
[2] https://github.com/yoseplee/vrf
[3] https://github.com/klauspost/reedsolomon
[4] https://github.com/amiller/HoneyBadgerBFT
[5] https://github.com/dedis/kyber
[6] https://github.com/bft-smart/library



(a) Basic SimpleFT with 7 replicas

(b) Basic SimpleFT with 31 replicas

(c) Layered SimpleFT with 7 replicas

(d) Layered SimpleFT with 31 replicas

Figure 7: Variation of the throughput when the number of Byzantine replicas (f) changes and the value of easiness (e) is adjusted

argument of easiness e to the SORTITION function and propose a simple easiness adjustment mechanism to change the probability of winning sortition, . In this section, we try to measure how can Byzantine replicas reduce the throughput and whether the easiness adjustment mechanism makes sense.

We conduct the experiments with 7 and 31 replicas, respectively. For a system with n replicas, we simply set the number of Byzantine replicas as $\lfloor n/6 \rfloor$ and model the actions of Byzantine replicas as being non-responsive [23]. To show the variation of throughput more clearly, we disable the automatic adjustment mechanism of e. Instead, we adjust the value of e manually. According to Equation 10, to eliminate the effects brought by $\lfloor n/6 \rfloor$ Byzantine replicas, e should be set as 1.2.

Experimental results are shown in Figure 7. By comparing two lines (blue and green lines) with $e = 1$ in each figure, we can find that Byzantine replicas can obviously reduce the system throughput. Particularly, in the experiment with 31 replicas and 8,000 transactions within a batch, Byzantine replicas reduce the throughput of layered SimpleFT by 23.3%, as shown in Figure 7d. On the other hand, when the value of e is increased from 1 to 1.2, the throughput gets improved. To be more specific, as shown by the red and blue lines in Figure 7a or Figure 7b, the system with $f = \lfloor n/6 \rfloor$ and $e = 1.2$ achieves a nearly identical performance to that with $f = 0$ and $e = 1$. Therefore, we can conclude that the adjustment mechanism of e is exactly effective in eliminating the bad effects

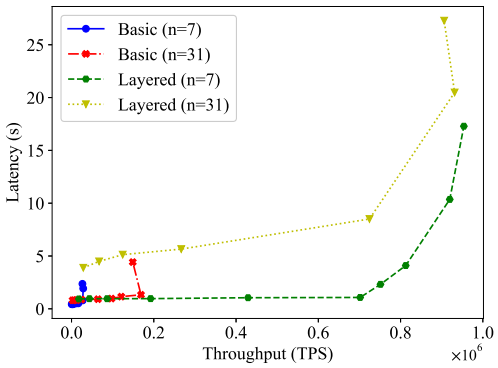


Figure 8: Latency vs. throughput

brought by Byzantine replicas, especially in basic SimpleFT. On the other hand, from Figure 7c or Figure 7d, we can find that there is still a gap between the blue and red lines. The reason for it can be explained as follows. Byzantine replicas will refuse to broadcast not only VBlocks but also DBlocks. Although the increase of e can lead the expected number of VBlocks in one round of sortition to remain as 1, the DBlocks in one epoch will be less than n. Therefore, the Byzantine replicas will slightly reduce the system throughput in layered SimpleFT, even though the value of e has been adjusted.

7.5 Trade-off between latency and throughput

Generally speaking, the throughput of a system is impossible to increase indefinitely. When the throughput reaches a threshold, not only the throughput cannot increase anymore, but also the latency will grow sharply. To avoid the system from delivering a fairly high latency, the throughput should be limited within a threshold. In other words, there is a trade-off between the latency and the throughput. In this section, we do an analysis of this trade-off. Particularly, we gradually increase the batch size to saturate the system throughput, during which process pairs of throughput and latency are recorded. Besides, all the replicas are set as correct ones.

Experimental results are shown in Figure 8. It is easy to find that layered SimpleFT has a higher upper bound of the throughput than basic SimpleFT. By comparing basic SimpleFT containing different numbers of replicas, we find a larger number of replicas can bring a higher upper bound, which demonstrates the good scalability of basic SimpleFT. On the other hand, layered SimpleFT with different numbers of replicas delivers a similar upper bound. The reason for it is that the layered architecture enables massive DBlocks to be propagated in parallel, which causes the network resources to become the bottleneck easily. Therefore, both the layered SimpleFT with $n = 7$ and $n = 31$ can only obtain the throughput limited by the network bandwidth, which is roughly equal to each other. Besides, we can find that the layered SimpleFT with $n = 31$ delivers a higher latency than the others. This is because the broadcast of VBlock data and vote messages may be delayed, due to the propagation of massive DBlocks. Maybe we can broadcast VBlock data and vote messages via a separate network link, which remains to be our future work.

8 RELATED WORK

According to different timing assumptions, the consensus protocols can be divided into three categories, namely synchronous consensus, partially-synchronous consensus, and asynchronous consensus. In this section, we make a summary of these consensus-related works.

8.1 Synchronous consensus

Originated from the oral-message and signed-message solutions proposed by Lamport et al. [30], quite a number of early consensus protocols are designed under the assumption of the synchronous networks, such as Rampart [41] and SecureRing [28]. Besides, some wonderful works in recent years also take the synchronous-network assumption, including Pili [15] and Sync HotStuff [1]. The greatest advantage of the synchronous consensus is the protocol's simplicity and conciseness, which can be understood and implemented with ease. However, as we have discussed in Section 2.1, the synchronous assumption can be easily violated in an unstable network environment, for example in a worldwide WAN deployment. The violation of the timing assumption may compromise the safety property, which causes the consensus protocol to fail.

Besides, if the timeout parameter Δ is set very large to maintain safety, the consensus efficiency can be dragged down to quite a low level. The most representative of this line of work is Bitcoin [37], which sets Δ (i.e., block interval) as ten minutes and can only process at most 7 *transactions per second* (TPS). Although Ethereum [45] makes an improvement to Bitcoin by reducing Δ to about fifteen seconds, it can only offer at most 15 TPS.

8.2 Partially-synchronous consensus

As the first one to do a combination of synchronous network and asynchronous network, Dwork et al. [18] pioneer the study of partially-synchronous consensus. The best-known work in this category is PBFT [12], which is considered the de facto standard of the consensus protocol. PBFT successfully reduces the protocol complexity from exponential level to polynomial level, which is adopted by a large number of blockchain systems, including Hyperledger Fabric [2] and ELASTICO [32]. Following PBFT, lots of wonderful works strive to improve the performance of PBFT, either by introducing the fast-path commitment mechanism (e.g., Zyzzyva [29] and SBFT [23]) or the trusted hardware (e.g., FastBFT [31] and Hybster [3]). Besides, with the emergence of blockchain technology, data structures of blocks and chains also bring new inspirations to the consensus design, which spawns a line of works, such as Tendermint [6], Pala [14], HotStuff [46], and Streamlet [13].

On the one hand, the partially-synchronous consensus can guarantee safety regardless of whether the network is synchronous or asynchronous. On the other hand, once the network becomes synchronous, the partially-synchronous consensus claims to promise the liveness property. However, as analyzed in [35] and [24], the partially-synchronous consensus is vulnerable to Byzantine replicas' attacks, making the consensus unusable or inefficient. To be more specific, an adversarial network scheduler can be constructed to generate network partitions when the correct replicas are leaders, and heal the partitions when the Byzantine replicas are leaders. In this regard, the system cannot reach a consensus at any time. What's worse, even if the partitions are healed finally with correct leaders, which satisfies the assumption of partial synchronization, it will take a long period to recover from the network partitions, resulting in low consensus efficiency.

8.3 Asynchronous consensus

Although the FLP theorem [19] proclaimed the impossibility of designing a deterministic consensus in the asynchronous network with even one faulty replica, a mass of research has been conducted to circumvent this impossibility result. Apart from making a stronger timing assumption as in the above sections, lots of works give up the determinism characteristic. In other words, almost all of the asynchronous consensus protocols introduce randomness in the protocol design, the early representatives of which include SINTRA [10] and CKPS01 [8]. Based on these early attempts, a range of recent works strives to improve the efficiency to make the asynchronous consensus practically usable, such as HoneybadgerBFT [35], BEAT [17], Aleph [20], and DAG-Rider [27].

Although these asynchronous protocols promise much stronger robustness, they are usually too difficult to understand and implement, especially for a non-expert learner. The more complicated a protocol is, the more bugs its implementation will suffer from [38]. By contrast, Bitcoin proposes a new path to design the consensus protocol, which is quite simple and understandable. Therefore, in our work (i.e., SimpleFT), we inherit the design path from Bitcoin to devise a simple asynchronous consensus. Besides, most of the

existing asynchronous protocols take the *Asynchronous Byzantine Agreement* (ABA) [5] algorithm as its core component. However, the ABA algorithm is subject to the liveness problem that the replicas involved in the ABA algorithm may not be able to terminate graciously.

9 CONCLUSION

Inspired by the simplicity of the Bitcoin protocol, we propose SimpleFT in this paper, which is a simple consensus protocol for the asynchronous network. Corresponding to the three phases disassembled from the Bitcoin protocol, we devise or introduce the sortition algorithm, reliable broadcast algorithm, and quorum certificate mechanism to implement the functions of block proposal, dissemination, and confirmation, respectively. Furthermore, to avoid the network resources from being wasted, we introduce the layered architecture to SimpleFT and divide the blocks into two types, namely vote blocks and data blocks. Multiple data blocks can be confirmed in one epoch, thus avoiding the waste of block propagation and increasing the system throughput. Comprehensive analysis validates SimpleFT's correctness and experimental results demonstrate its efficiency. We hope this work can open new ways of consensus design and bring some insights to others in the future.

REFERENCES

[1] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 106–118.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[3] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*. 222–237.

[4] Alexandra Boldyreva. 2003. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *International Workshop on Public Key Cryptography*. Springer, 31–46.

[5] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

[6] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph.D. Dissertation. University of Guelph.

[7] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

[8] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*. Springer, 524–541.

[9] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.

[10] Christian Cachin and Jonathan A Poritz. 2002. Secure intrusion-tolerant replication on the Internet. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 167–176.

[11] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 191–201.

[12] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[13] Benjamin Y Chan and Elaine Shi. 2020. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 1–11.

[14] TH Hubert Chan, Rafael Pass, and Elaine Shi. 2018. Pala: A simple partially synchronous blockchain. *Cryptology ePrint Archive* (2018).

[15] TH Hubert Chan, Rafael Pass, and Elaine Shi. 2018. Pili: An extremely simple synchronous blockchain. *Cryptology ePrint Archive* (2018).

[16] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults.. In *NSDI*, Vol. 9. 153–168.

[17] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2028–2041.

[18] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[19] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[20] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 214–228.

[21] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*. 51–68.

[22] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer systems*. 363–376.

[23] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.

[24] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 803–818.

[25] Markus Jakobsson and Ari Juels. 1999. Proofs of work and bread pudding protocols. In *Secure information networks*. Springer, 258–272.

[26] Márk Jelasity. 2011. *Gossip*. Springer Berlin Heidelberg, Berlin, Heidelberg, 139–162. https://doi.org/10.1007/978-3-642-17348-6_7

[27] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 165–175.

[28] Kim Potter Kihlstrom, Louise E Moser, and P Michael Melliar-Smith. 1998. The SecureRing protocols for securing group communication. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, Vol. 3. IEEE, 317–326.

[29] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[30] Leslie Lamport, Robert Shostak, and Marshall Pease. 2019. The Byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*. 203–226.

[31] Jian Liu, Wenting Li, Ghassan O Karame, and N Asokan. 2018. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Trans. Comput.* 68, 1 (2018), 139–151.

[32] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 17–30.

[33] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.

[34] Silvio Micali, Michael Rabin, and Salil Vadhan. 1999. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*. IEEE, 120–130.

[35] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 31–42.

[36] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. 2017. A review on consensus algorithm of blockchain. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, 2567–2572.

[37] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[38] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

[39] Rafael Pass and Elaine Shi. 2017. The sleepy model of consensus. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 380–409.

[40] Marshall Pease, Robert Shostak, and Leslie Lamport. 1980. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)* 27, 2 (1980), 228–234.

[41] Michael K Reiter. 1994. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. 68–80.

[42] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. 2020. Blockene: A High-throughput Blockchain Over Mobile Devices. In *14th {USENIX}*

Symposium on Operating Systems Design and Implementation ({OSDI} 20). 567–582.

[43] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[44] Joao Sousa, Alysson Bessani, and Marko Vukolic. 2018. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 51–58.

[45] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[46] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

A PSEUDOCODE

Algorithm 3 Dissemination of blocks in basic SimpleFT or VBlocks in layered SimpleFT (for p_i)

1: **Let** k represent the epoch number and ξ_B represent the precursor proof for B.

2: **function** BLKDISSEM$_i(k)$
3: **upon** receiving $\langle \texttt{BLK}, k, B_k \rangle$ from RBCBLOCK$_i$ **then**
4: send *true* to *sig* of MULRUDSORT$_i$
5: **return**
6: **upon** receiving $\langle b, x, \psi, r, \mu \rangle$ from MULRUDSORT$_i(k)$ **then**
7: $\pi_k \leftarrow \langle b, x, \psi, r, \mu, \xi_{B_{k-1}} \rangle$
8: **if** basic SimpleFT **then**
9: $B_k \leftarrow \langle b_k, H(B_{k-1}), \pi_k \rangle_i$
10: **else**
11: $B_k \leftarrow \langle H(B_{k-1}), \pi_k, \{\langle H(D_0), \sigma_0 \rangle, ..., \langle H(D_l), \sigma_l \rangle\} \rangle_i$
12: send $\langle \texttt{BLK}, k, B_k \rangle_i$ to RBCBLOCK$_i$
13: **return**
14: **end function**

Block dissemination algorithm: The process of block dissemination is described as Algorithm 3. A replica will wait for either a valid block proposal from the RBCBLOCK function or a result from the MULRUDSORT function. If the replica firstly receives a valid block from others, it will send the signal to the MULRUDSORT function to stop it (Lines 3-5). Otherwise, it will keep running the MULRUDSORT function to acquire the eligibility for the block proposal and then broadcast the new block, as shown in line 6. According to the type of SimpleFT, the replica will create a normal block (lines 8-9) or a VBlock (lines 10-11). Note that except the sortition proof ψ and round proof μ received from the MULRUDSORT function, the replica will also attach the precursor proof ξ for its parent block, as Line 7 shows.

RBC algorithm for normal blocks or VBlocks: The RBC algorithm for normal blocks in basic SimpleFT or VBlocks in layered SimpleFT is shown in Algorithm 4. In general, this algorithm mainly implements the ideas from Bracha's broadcast [5], which has also been described in HoneybadgerBFT [35] and Dumbo [24]. We present it here to make a comparison with the RBC algorithm for votes (Algorithm 5) or the PRBC algorithm for DBlocks (Algorithm 6). An RBC algorithm mainly consists of four steps:

Algorithm 4 Reliable broadcast of normal blocks in basic SimpleFT or VBlocks in layered SimpleFT (for p_i)

1: **Let** MERKLETREE, BRANCH, and DECODE represent the functions related to the merkle tree.

2: **function** RBCBLOCK$_i$( )
3: **upon** receiving $\langle \texttt{BLK}, k, B_k \rangle_i$ from BLKDISSEM$_i(k)$ **then**
4: $\{s_j \mid 1 \le j \le n\} \leftarrow$ ERASURECODE$(B_k, f+1, n)$
5: $T \leftarrow$ MERKLETREE$(\{s_j\})$; $r \leftarrow$ ROOT(T)
6: $\forall 1 \le j \le n, b_j \leftarrow$ BRANCH(T, j)
7: send $\langle \texttt{VAL\_BLK}, i, k, r, b_j, s_j \rangle_i$ to p_j
8: **upon** receiving $\langle \texttt{VAL\_BLK}, j, k, r, b_i, s_i \rangle_j$ from p_j **then**
9: **if** ! $blkValRecvd[j][k]$ **then**
10: $blkValRecvd[j][k] \leftarrow true$
11: BROADCAST$(\langle \texttt{ECHO_BLK}, j, k, r, b_i, s_i \rangle_i)$
12: **upon** receiving $\langle \texttt{ECHO\_BLK}, m, k, r, b_j, s_j \rangle_j$ from p_j **then**
13: $shards[m][k] += \langle \texttt{ECHO\_BLK}, m, k, r, b_j, s_j \rangle_j$
14: **if** $|shards[m][k]| = 2f+1$ **then**
15: BROADCAST$(\langle \texttt{RDY_BLK}, m, k, r \rangle_i)$
16: $blkRdyBc[m][k] \leftarrow true$
17: **upon** receiving $\langle \texttt{RDY\_BLK}, m, k, r \rangle_j$ from p_j **then**
18: $blkRdy[m][k] += \langle \texttt{RDY\_BLK}, m, k, r \rangle_j$
19: **if** $|blkRdy[m][k]| = f+1$ && ! $blkRdyBc[m][k]$ **then**
20: BROADCAST$(\langle \texttt{RDY_BLK}, m, k, r \rangle_i)$
21: $blkRdyBc[m][k] \leftarrow true$
22: **if** $|blkRdy[m][k]| = 2f+1$ **then**
23: $B_k \leftarrow$ DECODE$(shards, m, k)$
24: send $\langle \texttt{BLK}, k, B_k \rangle$ to Line 3 of BLKDISSEM$_i(k)$
25: send $\langle \texttt{BLK}, k, B_k \rangle$ to channel ch_i of RBCVOTE$_i(k)$
26: **end function**

- **Step 1 (lines 3-7):** The sender initiates a process of RBC by broadcasting the VAL messages. If the size of data is too large, the sender can make use of the erasure code mechanism and Merkle tree structure to reduce the network overhead.
- **Step 2 (lines 8-11):** Each replica broadcasts the ECHO messages once receiving a valid VAL message.
- **Step 3 (lines 12-16):** After receiving $2f+1$ or more ECHO messages, a replica will broadcast the RDY messages. Every received ECHO message will be cached in a set *shards*.
- **Step 4 (lines 17-25):** Once receiving $2f+1$ RDY messages, a replica can decode *shards* to restore the original data, which will be sent to the BLKDISSEM function in Algorithm 3 and RBCVOTE function in Algorithm 5.

We remark that some validation details, such as the validation of received Merkle branches, are omitted in Algorithm 4 to make the pseudocode more concise.

RBC algorithm for votes: The voting mechanism for normal blocks in basic SimpleFT is quite similar to votes for VBblocks in layered SimpleFT. The RBC algorithm for votes is described in Algorithm 5. Since the size of a vote message is quite small, neither the erasure-code mechanism nor the Merkle-tree structure is needed in Step 1, in comparison with Algorithm 4. Each replica can only vote for one block in basic SimpleFT (or VBlock in layered SimpleFT)

Algorithm 5 Reliable broadcast of votes (for p_i)

```
1:  function RBCVOTE_i( )
2:      upon receiving ⟨BLK, k, B_k⟩_i from ch_i then
3:          h ← hash(B_k)
4:          BROADCAST(⟨VAL_VOTE, k, h, i⟩_i)
5:      upon receiving ⟨VAL_VOTE, k, h, j⟩_j from p_j then
6:          if ! voteValRecvd[j][k] then
7:              voteValRecvd[j][k] = true
8:              BROADCAST(⟨ECHO_VOTE, k, h, j⟩_i)
9:      upon receiving ⟨ECHO_VOTE, k, h, m⟩_j from p_j then
10:         votes[m][k]+ = ⟨ECHO_VOTE, k, h, m⟩_j
11:         if |votes[m][k]| = 2f + 1 then
12:             BROADCAST(⟨RDY_VOTE, k, h, m⟩_i)
13:             voteRdyBc[m][k] = true
14:     upon receiving ⟨RDY_VOTE, k, h, m⟩_j from p_j then
15:         voteRdy[m][k]+ = ⟨RDY_VOTE, k, h, m⟩_j
16:         if |voteRdy[m][k]| = f+1 && ! voteRdyBc[m][k] then
17:             BROADCAST(⟨RDY_VOTE, m, k, r⟩_i)
18:             voteRdyBc[m][k] = true
19:         if |voteRdy[m][k]| = 2f + 1 then
20:             OUTPUT ⟨VOTE, k, h, m⟩
21: end function
```

in each epoch, which is checked by others, as Line 6 shows. In other words, if a Byzantine replica maliciously initiates RBC votes for multiple conflicting blocks in the same epoch, one of these votes cannot pass the checking in Line 6. Therefore, only one vote can be broadcast successfully. Besides, in the real implementation, multiple non-conflicting votes can be broadcast in one process of RBC, which can reduce the communication overhead.

PRBC algorithm for DBlocks: As discussed in Section 5.2, the broadcast of DBlocks must be attached with the proof. We borrow the idea from the Provable Reliable Broadcast (PRBC) algorithm in Dumbo [24] to broadcast DBlocks, which is described in Algorithm 6. The main body of Algorithm 6 resembles Algorithm 4 largely, except that Algorithm 6 adds an extra step to assemble the proof. To be more specific, after restoring the DBlock from *shards*, a replica will further broadcast a DONE message indicating that it has received the DBlock correctly. A threshold share signature on this DBlock will be included in the DONE message, as shown by Lines 24-25. After receiving $f + 1$ DONE messages, the replica can assemble the PRBC proof (σ) based on the threshold share signatures. Both the DBlock data and the PRBC proof will be stored in the DBlock pool, as Lines 29-30 show.

Recall that in Section 5.1, to avoid a Byzantine replica from broadcasting VBlocks arbitrarily, we add an extra validation mechanism to **Step 2** of Algorithm 6. As shown by line 9, only the DBlock with a sequence number *sn* that is no larger than $e + w$ can pass the validation. w is an empirical value, which can be adjusted according to the transaction workloads. The higher the workload is, the larger the value of w should be set as.

Algorithm 6 Provable reliable broadcast of DBlocks (for p_i)

Let SIGSHARE and COMBINE denote the functions relevant to the threshold signature. Let e be the epoch number of p_i's latest received VBlock and w represent an empirical value to control how fast blocks are proposed.

```
2:  function PRBC_i( )
3:      upon receiving ⟨DLK, txs, sn⟩_i from p_i itself then
4:          {s_j | 1 ≤ j ≤ n} ← ERASURECODE(txs, f + 1, n)
5:          T ← MERKLETREE({s_j}); r ← ROOT(T)
6:          ∀1 ≤ j ≤ n, b_j ← BRANCH(T, j)
7:          send ⟨VAL_DLK, i, sn, r, b_j, s_j⟩_i to p_j
8:      upon receiving ⟨VAL_DLK, j, sn, r, b_i, s_i⟩_j from p_j then
9:          if ! dlkValRecvd[j][sn] && sn ≤ e + w then
10:             dlkValRecvd[j][k] ← true
11:             BROADCAST(⟨ECHO_DLK, j, sn, r, b_i, s_i⟩_i)
12:     upon receiving ⟨ECHO_DLK, m, sn, r, b_j, s_j⟩_j from p_j then
13:         shards[m][sn] += ⟨ECHO_DLK, m, sn, r, b_j, s_j⟩_j
14:         if |shards[m][sn]| = 2f + 1 then
15:             BROADCAST(⟨RDY_DLK, m, sn, r⟩_i)
16:             dlkRdyBc[m][sn] ← true
17:     upon receiving ⟨RDY_DLK, m, sn, r⟩_j from p_j then
18:         dlkRdy[m][sn]+ = ⟨RDY_DLK, m, sn, r⟩_j
19:         if |dlkRdy[m][sn]| = f + 1 && ! dlkRdyBc[m][sn] then
20:             BROADCAST(⟨RDY_DLK, m, sn, r⟩_i)
21:             dlkRdyBc[m][sn] ← true
22:         if |dlkRdy[m][sn]| = 2f + 1 then
23:             txs ← DECODE(shards, m, sn)
24:             κ ← SIGSHARE_{f+1}(hash(txs))
25:             BROADCAST(⟨DONE_DLK, m, sn, κ⟩_i)
26:     upon receiving ⟨DONE_DLK, m, sn, κ⟩_j from p_j
27:         dlkDone[m][sn] += κ
28:         if |dlkDone[m][sn]| = f + 1
29:             σ ← COMBINE_{f+1}(dlkDone[m][sn])
30:             store ⟨DLK, txs, m, sn, σ⟩ to the local DBlock pool
31: end function
```