Byzantine Fault Tolerance in Partially Connected Asynchronous Networks

Yongge Wang
College of Computing and Informatics, UNC Charlotte
Charlotte, NC 28223, USA
yonwang@uncc.edu

February 10, 2020

Abstract

We review several widely deployed solutions for the Byzantine Fault Tolerance (BFT) problem and analyze their security in asynchronous networks. There are two types of widely accepted definitions for partial synchronous networks. In the Type I network, Denial of Service (DoS) attack is not allowed and in the Type II network, DoS attack is allowed before the Global Stabilization Time (GST). When DoS attack is allowed, the point-to-point communication channel and the broadcast channel are not reliable. We show that if either the broadcast channel or the point-to-point communication channel is not reliable (e.g., before GST) then several widely deployed BFT protocols (e.g., Tendermint BFT) would reach a deadlock before GST and the deadlock could not be removed after GST. To make things worse, we show that, for most of our attacks, the adversary only needs to control one participant to carry out the attack instead of controlling $\lfloor \frac{n-1}{3} \rfloor$ participants. Thus these BFT protocols are not secure in the Type II partial synchronous networks. Furthermore, in these protocols, if a participant does not receive appropriate messages within a fixed time period, it initiates a view change process. After a view change, participants will no long accept messages from previous views. Thus our attacks in Type II networks will also work against these protocols in Type I networks . Consequently, these protocols are not secure in any of the widely accepted partial synchronous networks. It should be noted that Tendermint BFT has been adopted in more than 40% Proof of Stake Blockchains such as Cosmos and Hyperledger burrow. Based on our analysis of BFT security requirements for partial synchronous networks, we propose a BFT protocol BDLS and prove its security in partial synchronous networks. It is shown that BDLS is one of the most efficient BFT protocols in partial synchronous networks. The BDLS protocol could be used in several application scenarios such as state machine replication or as blockchain finality gadgets.

1 Introduction

It is challenging to design consensus protocols for distributed computer system in an asynchronous environment. For non-malicious (non-Byzantine) faults, several practical protocols such as Paxos [14] and Raft [16] have been deployed in relatively closed environments. For example, Google, Microsoft, IBM, and Amazon have used Paxos in their storage or cluster management systems. Both Paxos and Raft can tolerate $\lfloor \frac{n-1}{2} \rfloor$ non-Byzantine faults.

Lamport, Shostak, and Pease [15] and Pease, Shostak, and Lamport [17] initiated the study of reaching consensus in face of Byzantine failures and designed the first synchronous solution for Byzantine agreement. Dolev and Strong [8] proposed an improved protocol in a synchronous network with $O(n^3)$ communication complexity. By assuming the existence of digital signature schemes and a public-key infrastructure, Katz and Koo [12] proposed an expected constant-round BFT protocol in a synchronous network setting against $\lfloor \frac{n-1}{2} \rfloor$ Byzantine faults.

For an asynchronous network, Fischer, Lynch, and Paterson [10] showed that there is no deterministic protocol for the BFT problem in face of a single failure. Their proof is based on a diagonalization construction and has two assumptions: (1) when a process writes a bit on the output register, it is finalized and can not change anymore; and (2) an honest process runs infinitely many steps in a run. Several researchers have tried to design BFT consensus protocols to circumvent the impossibility. For example, Ben-Or [1] initiated the probabilistic approach to BFT consensus protocols in completely asynchronous networks and Dwork, Lynch, and Stockmeyer [9] designed BFT consensus protocols in partial synchronous networks. Castro and Liskov [5] initiated the study of practical BFT consensus protocol design and introduced the PBFT protocol for partial synchronous networks. The core idea of PBFT has been used in the design of several widely adopted BFT systems such as Tendermint BFT [3]. Tendermint BFT has been used in more than 40% Proof of State blockchains (see, e.g., [13]) such as the "Internet of Blockchain" Cosmos [6]. More recently, Yin et al [22] improved the PBFT/Tendermint protocol by changing the mesh communication network in PBFT to hub-like (or star) communication networks in HotStuff and by using threshold cryptography. Facebook's Libra blockchain has adopted HotStuff in their LibraBFT protocol [19].

In the literature, there are mainly two kinds of partial synchronous networks for Byzantine Agreement protocols. In Type I partial synchronous networks, all messages are guaranteed to be delivered. In this type of networks, Denial of Service (DoS) attacks are not allowed and reliable point to point communication channels for all pairs of participants are required for the underlying networks. In Type II partial synchronous networks, the network becomes synchronous after an unknown Global Synchronization Time (GST). In this type of networks, Denial of Service (DoS) attacks are allowed before GST though it is not allowed after GST. The Type II network is more realistic and is commonly used in the literature.

Several partial synchronous network models for BFT design assume the existence of reliable broadcast communication channels for certain message transmission. In particular, these protocols normally leverage the gossip-based broadcast protocol in Bracha [2] which is based on the existence of reliable point-to-point communication channels for all pairs of participants. In particular, the broadcast protocol in Bracha [2] assumes a complete network to achieve "*a reliable message system in which no messages are lost or generated*". Since our Internet infrastructure is not a complete network, one needs to be very careful in building Internet based BFT protocols using Bracha's results. Specifically, one should not assume that there is a reliable broadcast channel before GST of Type II networks.

This paper shows that if there does not exist a reliable broadcast channel before GST of Type II networks, then one can launch attacks on several widely deployed BFT protocols (e.g., Tendermint BFT, and GRANDPA BFT [11]) so that participants would reach a deadlock before GST and the deadlock could not be removed after GST. Thus the participants would never reach an agreement and these BFT protocols are not secure in type II partial synchronous networks. For Type I networks, one does not know when the message could be delivered. Thus the broadcast protocol may be "unreliable" until the end of a fixed unknown time period. That is, the same attack in the Type II networks could be used to show that these protocol will reach deadlock before the end of this unknown time period. On the other hand, all these protocols will change views after certain timeout and after a view change, participants would not accept messages from previous views. That is, even all messages are delivered at the end of this unknown time period, participants discard these messages if they have changed views already. Thus these protocols will remain deadlocked. In a summary, our attacks show that these BFT protocols are insecure in all types of partial synchronous networks (including both Type I and Type II networks).

It should also be noted that though Tendermint [3] BFT protocol claims security in Type II asynchronous networks, it actually uses a Type I network model since it assumes a reliable point to point communication channel for each pair of participants in the network and no message is ever lost (including messages before GST). However, our discussion in the preceding paragraph shows that Tendermint is not secure in the Type I networks either. It should also be noted that in the first version of the LibraBFT specification (accessed on July 19, 2019, not accessible currently), its network model is a Type II partial synchronous network. In the current version [19] of the LibraBFT specification (dated as November 8, 2019 and accessed on February 9, 2020) , its network model is essentially a Type I partial synchronous network since all messages are delivered in the end (see pages 3 of Section 2 in [19]).

Based on the security requirement analysis for BFT protocols in asynchronous networks, we propose a BFT finality gadget protocol BDLS for blockchains. It should be noted that the first BFT protocol (i.e., the DLS protocol) for Type II networks was proposed by Dwork, Lynch, and Stockmeyer [9]. DLS protocol leverages a star network where participants only exchange messages via round leaders. The PBFT protocol allows all participants to broadcast their messages to all other participants. By leveraging this kind of mesh network, PBFT protocol was able to achieve consensus with reduced round complexity. By leveraging the lock-mechanisms in PBFT/Tendermint BFT protocols and changing the mesh network back to star network, HotStuff BFT/LibraBFT is able to achieve consensus with reduced communication complexity but increased round complexity. The BDLS protocol proposed in this paper is based on the original DLS protocol [9] and is able to achieve consensus with both reduced round complexity and reduced communication complexity. Specifically, BDLS has the same round complexity as PBFT and has reduced communication complexity than HotStuff BFT/LibraBFT. BDLS is proved to be secure in Type II partial synchronous networks and achieves the best performance among existing BFT protocols for blockchains. Though both BDLS and HotStuff BFT leverages star networks, BDLS employs the lock-mechanisms used in DLS protocol while HotStuff employs the lock-mechanisms used in PBFT/Temdermint BFT protocols. Thus BDLS could achieve consensus in 4 steps while HotStuff requires 7 steps to achieve consensus in synchrony.

The structure of the paper is as follows. Section 2 introduces various network models that we have interest in. Section 3 discusses several issues regarding broadcast channel reliability. Section 4 reviews the Tendermint BFT protocol and presents several attacks against it. Section 5 discusses several issues related to Ethereum's BFT finality gadgets. Section 6 reviews the Polkadot's GRANDPA BFT protocol and presents an attack against it. Section 7 presents the BDLS protocol design and analyzes its security within Type II networks.

2 Synchronous, asynchronous, and partial synchronous networks

Assume that the time is divided into discrete units called slots $T_0, T_1, T_2, \cdots$ where the length of the time slots are equal. Furthermore, we assume that: (1) the current time slot is determined by a publicly-known and monotonically increasing function of current time; and (2) each participant has access to the current time. In a synchronous network, if an honest participant P_1 sends a message m to a participant P_2 at the start of time slot T_i, the message m is guaranteed to arrive at P_2 at the end of time slot T_i. In the complete asynchronous network, the adversary can selectively delay, drop, or re-order any messages sent by honest parties. In other words, if an honest participant P_1 sends a message m to a participant P_2 at the start of time slot T_{i_1}, P_2 may never receive the message m or will receive the message m eventually at time T_{i_2} where $i_2 = i_1 + \Delta$. Dwork, Lynch, and Stockmeyer [9] considered the following two kinds of partial synchronous networks:

- Type I asynchronous network: $\Delta < \infty$ is unknown. That is, there exists a Δ but the participants do not know the exact value of Δ.
- Type II asynchronous network: $\Delta < \infty$ holds eventually. That is, the participant knows the value of Δ. But this Δ only holds after an unknown time slot $T = T_i$. Such a time T is called the Global Stabilization Time (GST).

For Type I asynchronous networks, the protocol designer supplies the consensus protocol first, then the adversary chooses her Δ. For Type II asynchronous networks, the adversary picks the Δ and the protocol designer (knowing Δ) supplies the consensus protocol, then the adversary chooses the GST. The definition of partial synchronous networks in [5, 22, 19] is the second type of partial synchronous networks. That is, the value of Δ is known but the value of GST is unknown. In such kind of networks, the adversary can selectively delay, drop, or re-order any messages sent by honest participants before an unknown time GST. But the network will become synchronous after GST. Several BFT protocols in the literature (e.g., Tendermint, GRANDPA, and the current version of LibraBFT dated on November 8, 2019) uses Type II networks, but they also assume that no message gets lost. With this additional assumption, the network is actually a Type I network since all messages are delivered within a time period GST+Δ where GST is unknown and Δ is known.

For the Type I network model, Denial of Service (DoS) attack is not allowed since message could be lost with DoS attacks. We think that it is more natural to use Type II asynchronous networks for distributed BFT protocol design and analysis. Thus this paper adopts the Type II network model unless specified otherwise.

3 Reliable broadcast communication channels

The difference between point-to-point communication channels and broadcast communication channels has been extensively studied in the literature. A reliable broadcast channel requires that the following two properties be satisfied.

1. Correctness: If an honest participant broadcasts a message m, then every honest participant accepts m.
2. Unforgeability: If an honest participant does not broadcast a message m, then no honest participant accepts m.

For complete networks, reliable broadcast protocols have been proposed in Bracha [2]. For a given integer k, a network is called k-connected if there exist k-node disjoint paths between any two nodes within the network. In non-complete networks, it is well known that $(2t + 1)$-connectivity is necessary for reliable communication against t Byzantine faults (see, e.g., Wang and Desmedt [21] and Desmedt-Wang-Burmester [7]). On the other hand, for broadcast communication channels, Wang and Desmedt [20] showed that there exists an efficient protocol to achieve probabilistically reliable and perfectly private communication against t Byzantine faults when the underlying communication network is $(t+1)$-connected. The crucial point to achieve these results is that: in a point-to-point channel, a malicious participant P_1 can send a message m_1 to participant P_2 and send a different message m_2 to participant P_3 though, in a broadcast channel, the malicious participant P_1 has to send the same message m to multiple participants including P_2 and P_3. If a malicious P_1 sends different messages to different participants in a reliable broadcast channel, it will be observed by its neighbors.

Though broadcast channels at physical layers are commonly used in local area networks, it is not trivial to design reliable broadcast channels over the Internet infrastructure since the Internet connectivity is not a complete graph and some direct communication paths between participants are missing (see, e.g., [15, 21]). Quite a few broadcast primitives have been proposed in the literature using message relays (see, e.g., Srikanth and Toueg [18], Bracha [2], and Dwork-Lynch-tockmeyer [9]). In the message relay based broadcast protocol, if an honest participant accepts a message signed by another participant, it relays the signed message to other participants. However, in order for these message relay based broadcast protocol to be reliable, it requires that the network graph is complete which is not true for the Internet environments.

A broadcast channel is *unreliable* if a malicious participant could broadcast a message m_1 to a proper subset of the participants but not to other participants. That is, some participants will receive the message m_1 while other participants will receive a different message m_2 or receive nothing at all. In next sections, we show that several BFT protocols are insecure due to the lack of reliable broadcast channels before GST (messages before GST could get lost or re-ordered by the definition). Thus it is important to design BFT protocols that could tolerate unreliable broadcast channels before GST.

In the following sections, if not specified explicitly, we will assume that there are $n = 3t + 1$ participants $P_0, \cdots, P_{n-1}$ for the BFT protocol and at least t of them are malicious. Furthermore, we assume that each participant has a public and private key pair where the public key is known to all participants. We use the notation $\langle\cdot\rangle_i$ to denote that the message is digitally signed by the participant P_i.

4 Security analysis of Tendermint BFT

Buchman, Kwon, and Milosevic [3] initiated the study of BFT protocols as a finality gadget for blockchains. Specifically, the authors in [3] proposed Tendermint BFT as an overlay atop a block proposal mechanism.

4.1 Tendermint BFT protocol

Tendermint BFT protocol [3] is based on the PBFT protocol. In Tendermint BFT, there are $n = 3t + 1$ participants $P_0, \cdots, P_{n-1}$ and at most t of them are malicious. Each participant maintains five variables `step`, `lockedV`, `lockedR`, `validV`, and `ValidR` throughout the protocol run. For each blockchain height h, the protocol runs from round to round until it reaches an agreement for the height h. Then the protocol moves to the next blockchain height. For each round, it contains three steps: *propose, prevote,* and *precommit*. For each height h, the participants start the process by initializing their five variables to: $\texttt{step} = \texttt{propose}$, $\texttt{lockedV} = \texttt{nil}$, $\texttt{lockedR} = -1$, $\texttt{validV} = \texttt{nil}$, and $\texttt{ValidR} = -1$. Then it starts from round 0 until an agreement is reached for the height h. There is a public function $proposer(h, r)$ that returns the round leader for a given round r of the height h. The round r of the height h proceeds as follows:

1. *propose*: The leader $P_i = proposer(h, r)$ distinguishes the two cases:
   - $r = 0$ or $\texttt{validV} = \texttt{nil}$: P_i chooses her proposal v and $vr = -1$.
   - $r > 0$ and $\texttt{validV} \neq \texttt{nil}$: P_i lets $v = \texttt{validV}$ and $vr = \texttt{ValidR}$

   P_i broadcasts the signed message
   $$\langle \text{PROPOSAL}, h, r, v, vr\rangle_i \tag{1}$$
   to all participants. All other participants P_j initialize the timeout counter to execute $\texttt{OnTimeoutPropose}(h, r)$.

2. *prevote*: For all participants P_j who are in $\texttt{step} = \texttt{propose}$, P_j distinguishes the following three cases:

- P_j receives (1) with $vr = -1$. If $\texttt{lockedR} = -1$ or $\texttt{validV} = v$, then P_j broadcasts the message $\langle \text{PREVOTE}, h, r, H(v)\rangle_j$. Otherwise, P_j broadcasts the message $\langle \text{PREVOTE}, h, r, \texttt{nil}\rangle_j$. P_j sets $\texttt{step} = \texttt{prevote}$.
- P_j receives (1) with $vr \geq 0$ and P_j has received $2t+1$ $\langle \text{PREVOTE}, h, vr, H(v)\rangle$. P_j distinguishes the following two cases
  - $\texttt{lockedR} \leq vr$ or $\texttt{lockedV} = v$: P_j broadcasts $\langle \text{PREVOTE}, h, r, H(v)\rangle_j$
  - Otherwise: P_j broadcasts the message $\langle \text{PREVOTE}, h, r, \texttt{nil}\rangle_j$.

  P_j sets $\texttt{step} = \texttt{prevote}$.
- P_j receives (1) with $vr \geq 0$ though P_j has not received $2t+1$ $\langle \text{PREVOTE}, h, vr, H(v)\rangle$. P_j does nothing.

3. *precommit*:
   (a) As soon as a participant P_j in step $\texttt{prevote}$ receives $2t+1$ messages $\langle \text{PREVOTE}, h, r, *\rangle$ for the first time, P_j initializes timeout counter to execute $\texttt{OnTimeoutPrevote}(h, r)$.
   (b) As soon as a participant P_j in step $\texttt{prevote}$ receives $2t+1$ messages $\langle \text{PREVOTE}, h, r, \texttt{nil}\rangle$ for the first time, P_j broadcasts $\langle \text{PRECOMMIT}, h, r, \texttt{nil}\rangle$ and sets $\texttt{step} = \texttt{precommit}$.
   (c) If P_j is in step $\texttt{prevote} \vee \texttt{precommit}$, has received the proposal (1), and has received $2t+1$ messages $\langle \text{PREVOTE}, h, r, H(v)\rangle$, then P_j carries out the following steps
   - If $\texttt{step} = \texttt{prevote}$, then P_j sets $\texttt{lockedV} = v$, $\texttt{lockedR} = r$, broadcasts $\langle \text{PRECOMMIT}, h, r, H(v)\rangle$, and sets $\texttt{step} = \texttt{precommit}$.
   - P_j sets $\texttt{validV} = v$ and $\texttt{validR} = r$.
4. *decision*: As soon as a participant P_j receives $2t+1$ messages $\langle \text{PRECOMMIT}, h, r, *\rangle$ for the first time, P_j initializes timeout counter to execute $\texttt{OnTimeoutPrecommit}(h, r)$. If P_j has not decided a value for the height h, has received the proposal (1), and has received $2t+1$ messages $\langle \text{PRECOMMIT}, h, r, H(v)\rangle$, then P_j sets v as the decision value for height h, resets values for the five variables, and goes to round 0 of height $h+1$.
5. *automatic update round*: During any time of the protocol, if a participant P_j receives $t+1$ messages for a round $r' > r$, P_j moves to round r'.
6. *Timeout functions*:
   (a) $\texttt{OnTimeoutPropose}(h, r)$: broadcast $\langle \text{PREVOTE}, h, r, \texttt{nil}\rangle$ and set $\texttt{step} = \texttt{prevote}$.
   (b) $\texttt{OnTimeoutPrevote}(h, r)$: broadcast $\langle \text{PRECOMMIT}, h, r, \texttt{nil}\rangle$ and set $\texttt{step} = \texttt{precommit}$.
   (c) $\texttt{OnTimeoutPrecommit}(h, r)$: move to round $r+1$ of height h.

4.2 Attacks on Tendermint BFT

In this section, we show that Tendermint BFT does not achieve the liveness property in partial synchronous networks. We describe our attack in the Type II networks where the broadcast channel is unreliable before GST. Specifically, we show that if a malicious participant could choose to broadcast a message to a subset of the users before GST, then the system will reach a deadlock and no new block will be created anymore (even after GST). In other words, the Tendermint BFT will reach deadlock before GST and the deadlock could not be removed after GST. We then extend these attacks on Tendermint BFT to Type I networks. For simplicity, we assume that for a given height h, the leader participant is P_0 and the participants in $\mathcal{P}_1 = \{P_0, \cdots, P_{t-1}\}$ are malicious. Furthermore, let $\mathcal{P}_2 = \{P_t, \cdots, P_{2t}\}$, and $\mathcal{P}_3 = \{P_{2t+1}, \cdots, P_{3t}\}$.

Attack 1. In round 0 of height h, P_0 chooses a minimal valid value v and broadcasts $\langle \text{PROPOSAL}, h, 0, v, -1\rangle$ to participants in $\mathcal{P}_1 \cup \mathcal{P}_2$. After receiving $\langle \text{PROPOSAL}, h, 0, v, -1\rangle$ from P_0, each participant $P_j \in \mathcal{P}_1$ broadcasts $\langle \text{PREVOTE}, h, 0, H(v)\rangle$ to participants in $\mathcal{P}_2$ and each participant $P_j \in \mathcal{P}_2$ broadcasts $\langle \text{PREVOTE}, h, 0, H(v)\rangle$ to all participants and sets $\texttt{step} = \texttt{prevote}$. Each participant $P_j \in \mathcal{P}_2$ receives $2t+1$ messages $\langle \text{PREVOTE}, h, 0, H(v)\rangle$. Thus the participant $P_j \in \mathcal{P}_2$ sets $\texttt{lockedV} = v$, $\texttt{lockedR} = 0$, $\texttt{step} = \texttt{precommit}$, $\texttt{validV} = v$, $\texttt{validR} = 0$, and then broadcasts $\langle \text{PRECOMMIT}, h, 0, H(v)\rangle$. Since each participant receives at most $t+1$ pre-commit messages for the value v, no decision will be made during the round 0. After timeout for round 0, all participants moves to round

1 of height h. The participants in $\mathcal{P}_1$ will become dormant from now on. If a participant in $\mathcal{P}_2$ becomes the leader of round 1, it will broadcast the proposal $\langle \text{PROPOSAL}, h, 1, v, 0\rangle$. Since participant P_j in $\mathcal{P}_3$ has received at most $t+1$ provote messages for the value v in round 0, P_j will do nothing until timeout. Thus no honest participant can collect sufficient prevote messages for v to move ahead. After timeout for round 1, the system will move to round 2 of height h. On the other hand, if a participant P_j in $\mathcal{P}_3$ becomes the leader of round 1, it will broadcast the proposal $\langle \text{PROPOSAL}, h, 1, v', -1\rangle$. Since P_0 has selected the value v as the minimal valid value and new transactions have been inserted into the system since then, the honest leader for round 1 will select a valid value $v' \neq v$ with high probability. Thus participants in $\mathcal{P}_2$ will not accept the proposal for v' and will broadcast $\langle \text{PROVOTE}, h, 1, \texttt{nil}\rangle$. That is, no agreement could be made during round 1 and the system will move to round 2 of height h after timeout. This process will continue forever without making an agreement for the height h even after GST.

Attack 2. One can launch an attack on Tendermint BFT so that some participants in $\mathcal{P}_2$ will decide on a value v for the height h (though no participant in $\mathcal{P}_3$ decides on any value for the height h) and then the system moves to the deadlock. It is noted that due to the lock function in Tendermint BFT and due to the blockchain property, the adversary will not be able to let the participants in $\mathcal{P}_3$ to decide on a different value for the height h or $h+1$.

In the preceding Attack 1, the malicious user needs to control t participants in the set $\mathcal{P}_1$. Indeed, we can revise the attack in such a way that the malicious user only needs to control one user P_0 to launch a similar attack. We use the same set $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$. But this time, we assume that only the leader P_0 is malicious and all other participants are honest.

Attack 3. In round 0 of height h, P_0 chooses a minimal valid value v and broadcasts $\langle \text{PROPOSAL}, h, 0, v, -1\rangle$ to participants in $\mathcal{P}_1 \cup \mathcal{P}_2$. P_0 then broadcasts $\langle \text{PREVOTE}, h, 0, H(v)\rangle$ to participants in $\mathcal{P}_1 \cup \mathcal{P}_2$ and becomes dormant. After receiving $\langle \text{PROPOSAL}, h, 0, v, -1\rangle$ from P_0, each participant $P_j \in (\mathcal{P}_1 \setminus \{P_0\}) \cup \mathcal{P}_2$ broadcasts $\langle \text{PREVOTE}, h, 0, H(v)\rangle$ to all participants and sets $\texttt{step} = \texttt{prevote}$. Each participant $P_j \in \mathcal{P}_1 \cup \mathcal{P}_2$ receives $2t+1$ messages $\langle \text{PREVOTE}, h, 0, H(v)\rangle$. The participant $P_j \in (\mathcal{P}_1 \setminus \{P_0\}) \cup \mathcal{P}_2$ sets $\texttt{lockedV} = v$, $\texttt{lockedR} = 0$, $\texttt{step} = \texttt{precommit}$, $\texttt{validV} = v$, $\texttt{validR} = 0$, and broadcasts $\langle \text{PRECOMMIT}, h, 0, H(v)\rangle$. Since each participant receives at most $2t$ pre-commit messages for the value v, no decision will be made during the round 0. A similar argument as in the Attack 1 can be used to show that the protocol will enter a deadlock. Please note in this Attack 3, participant P_j in $\mathcal{P}_3$ has received at most $2t$ prevote messages for the value v in round 0, which is still insufficient for P_j to accept a proposal for a locked value v from other participants.

5 Casper FFG

Buterin and Griffith [4] proposed the BFT protocol Casper the Friendly Finality Gadget (Casper FFG) as an overlay atop a block proposal mechanism. In Casper FFG, weighted participants validate and finalize blocks that are proposed by an existing proof of work chain or other mechanisms. To simplify our discussion, we assume that there are $n = 3t+1$ validators of equal weight. The Casper FFG works on the checkpoint tree that only contains blocks of height $100 * k$ in the underlying block tree. Each validator P_i can broadcast a signed vote $\langle P_i : s, t\rangle$ where s and t are two checkpoints and s is an ancestor of t on the checkpoint tree. For two checkpoints a and b, we say that $a \to b$ is a supermajority link if there are at least $2t+1$ votes for the pair. A checkpoint a is justified if there are supermajority links $a_0 \to a_1 \to \cdots \to a$ where a_0 is the root. A checkpoint a is finalized if there are supermajority links $a_0 \to a_1 \to \cdots \to a_i \to a$ where a_0 is the root and a is the direct son of a_i. In Casper FFG, an honest validator P_i should not publish two distinct votes

$$\langle P_i : s_1, t_1\rangle \quad \text{AND} \quad \langle P_i : s_2, t_2\rangle$$

such that either

$$h(t_1) = h(t_2) \quad \text{OR} \quad h(s_1) < h(s_2) < h(t_2) < h(t_1)$$

where $h(\cdot)$ denotes the height of the node on the checkpoint tree. Otherwise, the validator's deposit will be slashed. Casper FFG is proved to achieve accountable safety and plausible liveness in [4] where

1. achieve accountable safety means that two conflicting checkpoints cannot both be finalized (assuming that there are at most t malicious validators), and

2. plausible liveness means that supermajority links can always be added to produce new finalized checkpoints, provided there exist children extending the finalized chain.

In order to achieve he liveness property, [4] proposed to use the "correct by construction" fork choice rule: the underlying block proposal mechanism should "*follow the chain containing the justified checkpoint of the greatest height*".

The authors in [4] proposed to defeat the long-range revision attacks by a fork choice rule to never revert a finalized block, as well as an expectation that each client will "log on" and gain a complete up-to-date view of the chain at some regular frequency (e.g., once per month). In order to defeat the catastrophic crashes where more than t validators crash-fail at the same time (i.e., they are no longer connected to the network due to a network partition, computer failure, or the validators themselves are malicious), the authors in [4] proposed to slowly drains the deposit of any validator that does not vote for checkpoints, until eventually its deposit sizes decrease low enough that the validators who are voting are a supermajority. Related mechanism to recover from related scenarios such as network partition is considered an open problem in [4].

No specific network model is provided in [4]. Thus it is important to investigate the security of Casper FFG in various network models. The specification in [4] does not have sufficient details to guarantee its claimed plausible liveness. The authors mentioned that the Casper FFG could be used on top of most proof of work chains. However, without further restrictions on the block generation mechanisms, Casper FFG can reach deadlock (so plausible liveness property will not be satisfied). Assume that, at time T, the checkpoint a is finalized (where there is a supermajority link from a to its direct child b) and no vote for b's descendant checkpoint has been broadcast by any validator yet. Now assume that the underlying block production mechanism produced a fork starting from b. That is, b has two descendant checkpoints c and d. If t honest validators vote for c, $t+1$ honest validators vote for d, and t malicious validators vote randomly, then we reach a deadlock (since no link from b to its descendant can have a supermajority). If the checkpoints are 100 blocks away from each other and if it is expensive/slow to generate blocks (e.g., using PoW) then this kind of fork may be hard to happen though there is still a possibility.

6 Another finality gadget: Polkadot's GRANDPA

Based on the Casper FFG protocol, the project Polkadot (`https://wiki.polkadot.network/`) proposed a new BFT finality gadget protocol GRANDPA [11]. Specifically, Polkadot implements a nominated proof-of-stake (NPoS) system. At certain time period, the system elects a group of validators to serve for block production and the finality gadget. Nominators also stake their tokens as a guarantee of good behavior, and this stake gets slashed whenever their nominated validators deviate from their protocol. On the other hand, nominators also get paid when their nominated validators play by the rules. Elected validators get equal voting power in the consensus protocol. Polkadot uses BABE as its block production mechanism and GRANDPA as its BFT finality gadget. Here we are interested in the finality gadget GRANDPA (GHOST-based Recursive ANcestor Deriving Prefix Agreement) that is implemented for the Polkadot relay chain. GRANDPA contain two protocols, the first protocol works in partially synchronous networks and tolerates 1/3 Byzantine participants. The second protocol works in full asynchronous networks (requiring a common random coin) and tolerates 1/5 Byzantine participants. In contrast to Casper FFG, GRANDPA voters can cast votes simultaneously for blocks at different heights and GRANDPA only depends on finalized blocks to affect the fork-choice rule of the underlying block production mechanism.

The first GRANDPA protocol assumes that after an unknown time GST, the network becomes synchronous. However, it also assumes that all messages are delivered before time GST$+\Delta$ for some given value Δ. That is, no message gets lost. This network model is equivalent to our Type I asynchronous network and will not tolerate DoS attacks and network partition attacks. In the following paragraphs, we will show that GRANDPA is not even secure in the synchronous network.

Assume that there are $n = 3t+1$ participants $P_0, \cdots, P_{n-1}$ and at most t of them are malicious. Each participant stores a tree of blocks produced by the block production mechanism with the genesis block as the root. A participant can vote for a block on the tree by digitally signing it. For a set S of votes, a participant P_i equivocates in S if P_i has more than one vote in S. S is called tolerant if at most t participants equivocate in S. A vote set S has supermajority for a block B if

$$|\{P_i : P_i \text{ votes for } B*\} \cup \{P_i : P_i \text{ equivocates}\}| \geq 2t+1$$

where P_i votes for $B*$ mean that P_i votes for B or votes for a descendant of B. The 2/3-GHOST function $g(S)$ returns the block B of the maximal height such that S has a supermajority for B. If a tolerant vote set S has a supermajority for a block B, then there are at least $t+1$ voters who do vote for B or its descendant but do not equivocate. Based on this observation, it is easy to check that if $s \subseteq T$ and T is tolerant, then $g(S)$ is an ancestor of $g(T)$.

The authors in [11] defined the following concept of *possibility* for a vote set to have a supermajority for a block: "We say that it is *impossible* for a set S to have a supermajority for a block B if at least $2t+1$ voters either equicovate or vote for blocks who are not descendant of B. Otherwise it is *possible* for S to have a supermajority for B." Then the authors [11] claimed that "a vote set S is possible to have a supermajority for a block B if and only if there exists a tolerant vote set $T \supseteq S$ such that T has a supermajority for B". **Unfortunately**, this claim has semantic issues in practice. For example, assume that blocks B and C are inconsistent and the vote set S contains the following votes:

1. t malicious voters vote for B, one honest voter votes for B.
2. $2t$ honest voters vote for C.

By the definition of [11], S is not impossible to have a supermajority for B. Thus S is possible to have a supermajority for a block B. Since honest voters will not equivocate, there does not exist a semantically valid tolerant vote set $T \supseteq S$ such that T has a supermajority for B. This observation could easily be used to show that the GRANDPA protocol cannot achieve the liveness property (see our discussion in next paragraphs).

6.1 GRANDPA protocol

The GRANDPA protocol starts from round 1. For each round, one participant is designated as the primary and all participants know who is the primary. Each round consists of two phases: *prevote* and *precommit*. Let $V_{r,i}$ and $C_{r,i}$ be the sets of prevotes and precommits received by P_i during round r respectively. Let $E_{0,i}$ be the genesis block and $E_{r,i}$ be the last ancestor block of $g(V_{r,i})$ that is possible for $C_{r,i}$ to have a supermajority. If either $E_{r,i} < g(V_{r,i})$ or it is impossible for $C_{r,i}$ to have a supermajority for any children of $g(V_{r,i})$, then we say that P_i sees that round r is completable. Let Δ be a time bound such that it suffices to send messages and gossip them to everyone. The protocol proceeds as follows.

1. P_i starts round $r > 1$ if round $r-1$ is completable and P_i has cast votes in all previous rounds. Let $t_{r,i}$ be the time P_i starts round r.
2. If P_i is the primary of round r and has not finalized $E_{r-1,i}$, then it broadcasts $E_{r-1,i}$.
3. P_i waits until either it is at least time $t_{r,i} + 2\Delta$ or round r is completable. P_i *prevotes* for the head of the best chain containing $E_{r-1,i}$ unless P_i receives a block B from the primary with $g(V_{r-1}, i) \geq B > E_{r-1,i}$. In this case, P_i uses the best chain containing B.
4. P_i waits until $g(V_{r,i}) \geq E_{r-1,i}$ and one of the following conditions holds
   (a) it is at least time $t_{r,i} + 4\Delta$
   (b) round r is completable
   (c) it is impossible for $V_{r,i}$ to have a supermajority for any child of $g(V_{r,i})$ (this is an optional condition)

   Then P_i broadcasts a precommit for $g(V_{r,i})$

At any time after the precommit step of round r, if P_i sees that $B = g(C_{r,i})$ is descendant of the last finalized block and $V_{r,i}$ has a supermajority, then P_i finalizes B.

6.2 Attacks on GRANDPA

In this section, we show that GRANDPA protocol cannot achieve the liveness property even in the synchronous networks. Assume that $E_{r-1,0} = \cdots = E_{r-1,n-1}$. During round r, the block production mechanisms produced a fork for $E_{r-1,0}$. That is, two child blocks B and C of $E_{r-1,0}$ are produced. At round r, $t+1$ voters (including all malicious voters) prevote for B and the remaining honest $2t$ voters prevote for C. For each voter P_i, we have $g(V_{r,i}) = E_{r-1,i}$. Thus each P_i precommits $g(V_{r,i}) = E_{r-1,i}$. Now each voter P_i estimates $E_{r,i} = g(V_{r,i}) = E_{r-1,i}$. Since it is possible for $C_{r,i}$ to have a supermajority for any child of $E_{r,i}$, the round r is not completable. That is, the process stuck at round r forever.

Even if one can revise the “possible” definition in the GRANDPA to resolve the issues that we have discussed in the preceding paragraph, our attacks on Tendermint could be easily mounted against GRANDPA protocol also. Thus GRANDPA protocol could not be secure in Type II networks.

7 A secure BFT protocol in Type II partial synchronous networks

In this section, we propose a Byzantine Agreement Protocol that achieves safety and liveness properties in Type II partial synchronous networks. Though our protocol could be used in other scenarios such as State Machine Replication (SMR), we present the protocol as a finality gadget for blockchains. Assume that there is a separate block proposal mechanism that produces children blocks for finalized blocks by our BFT finality gadget. Let $B^0, \cdots, B^{h-1}$ be the blockchain where B^0 is the genesis block and B^{h-1} is the most recently finalized head block. The block proposal mechanism may produce several child blocks $B^h_0, B^h_1, \cdots, B^h_{n_0-1}$ of the current head block B^{h-1}. These child blocks are strictly ordered. For example, in proof of stake blockchain applications, each participant has a stake value for the chain height h and these child blocks may be ordered using proposer's stake values. However, it is beyond the scope of this paper to specify how these child blocks are ordered for general blockchains. It is the task for the BFT finality gadget to select the maximal block among these candidate child blocks as the next block B^h. Though the goal of the BFT protocol is to select the maximal child block as the final version of block B^h, this may not be true in certain scenarios. For example, if $t+1$ honest participants have seen the child block $B^h_{n_0-2}$ and have not seen the maximal block $B^h_{n_0-1}$ at the start of the protocol (at the same time, we may assume that the other t honest participants have seen the maximal block $B^h_{n_0-1}$), then our BFT protocol BDLS will finalize $B^h_{n_0-2}$ instead of $B^h_{n_0-1}$ (assuming that the t malicious participants submit the block $B^h_{n_0-2}$ to the leader). Secondly, our BFT protocol leverages the fact that a candidate block is self-certified. That is, the validity of a candidate child block can be verified by using the information contained in the candidate block itself against the currently finalized blockchain.

7.1 The BFT protocol BDLS

Our BFT protocol is based on the original DLS protocol in Dwork, Lynch, and Stockmeyer [9] and we call it a Blockchain version of DLS (BDLS). For each blockchain height h, BDLS protocol runs from round to round until it reaches an agreement for the height h. Then the protocol moves to the next blockchain height $h+1$. Let $P_0, \cdots, P_{n-1}$ be the $n = 3t+1$ participants of the protocol. Assume that there are n_0 valid candidate proposals $B^h_0 < B^h_1 < \cdots < B^h_{n_0-1}$ for the block B^h. During the protocol run, each participant P_i maintains a local variable $\text{BLOCK}_i \subseteq \{B^h_0, B^h_1, \cdots, B^h_{n_0-1}\}$ that contains the candidate blocks that it has learned so far. Participant P_i prefers the maximal block in BLOCK_i to be selected as the final block for B^h. The goal of the BDLS protocol is for participants $P_0, \cdots, P_{n-1}$ to reach a consensus on the finalized block B^h.

Generally, we can use a robust threshold signature scheme to reduce the authenticator complexity. For simplicity, the following protocol description is based on a standard digital signature scheme. It could be easily revised to used a threshold signature scheme. Following Dwork, Lynch, and Stockmeyer [9], we assume that all messages after the unknown GST (Global Stabilization Time) will be delivered in the same round and messages before round GST could get lost or re-ordered. Furthermore, though all participants have a common numbering for the round, they do not know when the round GST occurs. A candidate block B' is acceptable to P_i if P_i does not have a lock on any value except possibly B'. There is a public function $leader(h, r)$ that returns the round leader for a given round r of the height h. For each height h, the BDLS protocol proceeds from round to round (starting from round 0) until the participant decides on a value. The round r of the height h starts when at least $2t+1$ participants submit a round-change message to the leader participant. Specifically, the round r proceeds as follows where $P_i = leader(h, r)$ is the leader for round r:

1. Each participant P_j (including P_i) sends the signed message $\langle h, r, B'_j\rangle_j$ to the leader P_i where $B'_j \in \text{BLOCK}_j$ is the maximal acceptable candidate block for P_j. The message $\langle h, r, B'_j\rangle_j$ is considered as a round-change message. P_j then sets the timeout counter for the round r of height h.

2. If P_i receives at least $2t$ round-change messages (excluding himself), P_i sends himself a message as in Step 1 (if he has not done so yet) and starts the round r of height h. In these round-change messages, if there are at least

$2t+1$ signed messages from $2t+1$ participants with the same candidate block B' in step 1, then P_i broadcasts the following signed message (2) to all participants

$$\langle \text{lock}, h, r, B', \text{proof}\rangle_i \tag{2}$$

where proof is a list of at least $2t+1$ signed messages showing that B' is the candidate blocks for at least $2t+1$ participants (the proof also shows that round-change request has been authorized by at least $2t+1$ participants). If P_i does not receive such a block B', then P_i adds all received candidate blocks to its local variable BLOCK_j and broadcasts $\langle B'', \text{proof}\rangle$ where B'' is the candidate block $B'' = \max\{B : B \in \text{BLOCK}_i\}$ and proof is a list of at least $2t+1$ round-change messages from Step 1. P_i sets the timeout counter for the round r of height h if he has not done so yet.

3. If a participant P_j receives a valid $\langle B'', \text{proof}\rangle$ from P_i during Step 2, then it adds B'' to its BLOCK_j. If a participant P_j receives a valid message $\langle \text{lock}, h, r, B', \text{proof}\rangle_i$ from P_i in Step 2, then it does the following:
   (a) releases any potential lock on B' from previous round, but does not release locks on any other potential candidate blocks
   (b) locks the candidate block B' by recording the valid lock (2)
   (c) sends the following signed commit message to the leader P_i.

$$\langle \text{commit}, h, r, B'\rangle_j. \tag{3}$$

4. If P_i receives at least $2t+1$ commit messages (3), then P_i *decides* on the value B' and broadcasts the following decide message to all participants

$$\langle \text{decide}, h, r, B', \text{proof}\rangle_i. \tag{4}$$

   where proof is a list of at least $2t+1$ commit messages (3).

5. If a participant P_j receives a decide message (4) from step 4 or from its neighbor, it decides on the block B' for B^h and moves to the next height $h+1$ (that is, run the Step 1 of height $h+1$ by sending the round-change message). At the same time, the participant P_j propagates (broadcasts) the decide message (4) to all of its neighbors if it has not done so yet (see the following Remark 3 for more details on this). Otherwise, it goes to the following lock-release step:
   - (*lock-release*) If a participant P_j has some locked values, it broadcasts all of its locked values with proofs. A participant releases its lock on a value $\langle \text{lock}, h, r'', B'', \text{proof}\rangle_{i''}$ if it receives a lock $\langle \text{lock}, h, r', B', \text{proof}\rangle_{i'}$ with $r' \geq r''$ and $B' \neq B''$.
   - Move to the next round $r+1$ (i.e., run the Step 1 of height h with $r+1$).

6. *height synchronization*: If P_j receives a finalized bock of height h (e.g., a decide message (4)), P_j decides for height h and moves to height $h+1$.

7. *round synchronization*: If P_j receives a valid message from the leader for a round $r' > r$, P_j moves to round r'.

8. *timeout*: If P_j does not receive enough messages to move forward within a pre-fixed time period, it moves to the next round $r+1$ (i.e., run the Step 1 of height h with $r+1$). To make round synchronization more efficient, BDLS also recommends the use of PaceMaker (see Section 8).

Remark 1: In the BDLS protocol, the lock-release step is a mesh network broadcast. In some applications, one may prefer a star network to reduce the total number of messages from n^2 to n. One may achieve this kind of needs by replacing the "lock-release" step with the following additions to the protocol. At the Step 1 of round r, each participant P_j sends the message

$$\text{all-locked-values}, \langle h, r, B'_j\rangle_j$$

instead of only sending the message $\langle h, r, B'_j\rangle_j$ to P_i, where "all-locked-values" is the set of candidate blocks that P_j has locks on. During Step 2, if P_i cannot lock a candidate block during round r, then it broadcasts the candidate block

$B'' = \max\{B : B \in \text{BLOCK}_i\}$ together with all locked candidate blocks by all participants. It is straightforward to check that our security analysis in the next section remains unchanged for this protocol revision.
Remark 2: During Step 5 of the BDLS protocol, when a participant receives a `decide` message, it propagates/broadcasts the `decide` message to its neighbors. It is recommended that each participant keep broadcasting the signed `decide` message for height h regularly until it receives at least $2t$ broadcasts of the `decide` message for height h from other $2t$ participants. The importance of this propagation/broadcast is illustrated in Section 9.

7.2 Liveness and Safety

The security of BDLS protocol is proved by establishing a series of Lemmas. The proofs for Lemmas 7.1, 7.2, 7.3 and Theorem 7.4 follow from straightforward modifications of the corresponding Lemmas/Theorem in [9]. For completeness, we include these proofs here also.

Lemma 7.1 *It is impossible for two candidate blocks B' and B'' to get locked in the same round r of height h.*

Proof. In order for two blocks B' and B'' to get locked in one round r of height h, the leader $P_i = leader(h, r)$ must send two conflict lock messages (2) with different proofs. This can only happen if there exist at least $t + 1$ participants P_j each of whom equivocates two messages $\langle h, r, B' \rangle_j$ and $\langle h, r, B'' \rangle_j$ to P_i. This is impossible since there are at most t malicious participants. □

Lemma 7.2 *If the leader P_i decides a block value B' at round r of height h and r is the smallest round at which a decision is made. Then at least $t + 1$ honest participants lock the candidate block B' at round r. Furthermore, each of the honest participants that locks B' at round r will always have a lock on B' for round $r' \geq r$.*

Proof. In order for P_i to decide on B', at least $2t + 1$ participants send commit messages (3) to P_i at round r of height h. Thus at least $t + 1$ honest participants have locks on B' at round r. Assume that the second conclusion is false. Let $r' > r$ be the first round that the lock on B' is released. In this case, the lock is released during the lock-release step of round r' if some participant has a lock on another block $B'' \neq B'$ with associated round r'' where $r' \geq r'' \geq r$. Lemma 7.1 shows that it is impossible for a participant to have a lock on B'' ar round r. Thus the participant acquired the lock on B'' in round r'' with $r' \geq r'' > r$. This implies that, at the step 1 of round r'', more than $2t+1$ participants send signed messages $\langle h, r'', B'' \rangle$ to the leader participant. That is, at least $2t + 1$ participants have not locked B' at the step 1 of round r''. This contradicts the fact that at least $t + 1$ participants have locked B' at the start of round r''. □

Lemma 7.3 *Immediately after any lock-release step at or after the round GST, the set of candidate blocks locked by honest participants contains at most one value.*

Proof. This follows from the lock-release step. □

Theorem 7.4 *(Safety) Assume that there are at most t malicious participants. It is impossible for two participants to decide on different block values.*

Proof. Suppose that an honest participant P_i decides on B at round r and this is the smallest round at which the decision is made. Lemma 7.2 implies that at least $t + 1$ participants will lock B' in all future rounds. Consequently, no other block values other than B' will be acceptable to $2t + 1$ participants. Thus no participants will decide on any other values than B'. □

Theorem 7.5 *(Liveness) Assume that there are at most t malicious participants and valid candidate child blocks for B^h are always produced by the block proposal mechanism before the start of first round for height h for all h. Then BDLS protocol will finalize blocks for each height h. That is, the BDLS protocol will not reach a deadlock.*

Proof. We consider two cases. For the first case, assume that no decision has been made by any honest participants and no honest participant locks a candidate block at round r where $r \geq$ GST is the first round after GST that the leader participant is honest. In this case, if P_i receives $2t + 1$ signed messages for a candidate block B' in step 1 of round r, then all honest participants will decides on B' by the end of round r. Otherwise, P_i broadcasts the maximal candidate block B'' during step 2 of round r. Thus all honest participants will receive this maximum block and this candidate

Table 1: Comparison of BFT protocols with honest leader after GST

Steps	**PBFT**	**Tendermint BFT**	**HotStuff BFT**	**BDLS**
1	📡	📡	📡	👂
2	👋	👋	👂	📡
3	👋	👋	📡	👂
4			👂	📡
5			📡	
6			👂	
7			📡	
message complexity	$2n^2+n$	$2n^2+n$	$7n$	$4n$
authenticator complexity [22]	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$

block becomes the maximum acceptable candidate block for all honest participants. Then, in round $r' > r$ where r' is the smallest round after r that the leader participant is honest, all honest participants decide on a maximal block.

For the second case, assume that no candidate block is locked at the start of round GST and some participants hold a lock on a candidate block B'. By Lemma 7.3, there are at most one value locked by honest participants at the end of round GST. Furthermore, at the end of round GST, all the honest participants either decide on B' or obtain a lock on B'. Thus if no decision is made during round GST, the decision will be made during round GST$+1$. □

7.3 Performance comparison

In this section, we compare the performance of PBFT, Tendermint BFT, HotStuff BFT and our BDLS protocols. Three kinds of primitives are used in these protocol design: (1) broadcast from the leader to all participants; (2) all participants send messages to the leader; and (3) all participants broadcast. We use the following symbols to denote these primitives.

- 📡: leader broadcasts
- 👂: all participants send messages to the leader
- 👋: all participants broadcast

In the following, we compare the performance of these protocols after the network is synchronized (that is, after GST) and when the round has an honest leader. For all of these protocols, they will reach agreement within one run of the protocol assuming all participants have all the necessary input values at the start of the protocol and the leader is honest. Table 1 lists the steps of one run of these protocols. Furthermore, for BDLS, we use the approaches discussed in the Remarks after the BDLS protocol description to embed the lock-release step into Steps 1 and 2. For each 📡 or 👂 step, there is a total of n messages communicated in the network. For each 👋 step, there is a total of n^2 messages communicated in the network. The row "message complexity" of Table 1 lists the total number of messages communicated in the network for each run of the protocol. That is, in the ideal synchronized network, this is the total number of messages that are needed to achieve a consensus. These numbers show that BDLS has the smallest number of messages for a consensus in the synchronized network. Another way to compare the performance of BFT protocols is to compare the number of authenticator operations (signing and verifying) that are needed to achieve a consensus (see, e.g., [22]). Assume that all these schemes (except PBFT) use threshold digital signature schemes, then the row "authenticator complexity" of Table 1 lists the total number authenticator operations needed for each run of the protocol.

8 Chained BDLS and other implementation related issues

In order to improve efficiency, several blockchain BFT protocols (e.g., Ethereum Casper FFG, HotStuff BFT, and LibraBFT) adopt the chaining paradigm where the BFT protocol phases for commitment are spread across rounds.

That is, every phase is carried out in a round and contains a new proposal. The same techniques could be used to construct a chained BDLS. As noted in HotStuff BFT and LibraBFT, the block tree in chained LibraBFT and chained HotStuff BFT may contain "chains" that have gaps in round numbers. Thus the commit logic for LibraBFT and HotStuff BFT requires a 3-chain with contiguous round numbers whose last descendant has been certified. Since BDLS is a 2-phase BFT protocol, chained BDLS "decide" logic requires a 2-chain with contiguous round numbers whose last descendant has been certified.

For chained BFT protocol implementation, the BFT protocol participants for various rounds/heights should be relatively static. If the BFT protocol participants change from rounds to rounds or from heights to heights, it is not realistic to implement chained BFT protocols. Thus chained BFT protocol implementation is suitable for permissioned blockchains such as Libra blockchain while it is not suitable for permissionless blockchains where BFT protocol participants change frequently. The same rule applies to threshold digital signature scheme implementation for BFT protocols. That is, for permissionless blockchains where BFT protocol participants change frequently, it may have limited advantage in using threshold digital signature schemes since the expensive key set-up process has to be run each time when the participants set changes.

In most distributed BFT protocols, when the participants could not reach an agreement in one round, participants move to a new round by submitting round-change request. Thus BFT participants may be in different states and receive different messages. It is important to maximize the period of time when at least $2t+1$ honest participants are in the same round. PBFT protocol achieves round synchronization by exponentially increasing the timeout length for each round. That is, if the round 0 of height h has a timeout length of Δ, then the round r of height h will have a timeout length of $2^r\Delta$. On the other hand, Tendermint BFT achieves round synchronization by linearly increasing the timeout length for each round. That is, the round r has a timeout length of $r\Delta$ where Δ is the timeout length for round 0 of height h. HotStuff proposes a functionality called PaceMaker to achieve round synchronization without details on how to implement the PaceMaker. LibraBFT implemented the PaceMaker functionality in the following way. When a participant gives up on a certain round r, it broadcasts a timeout message carrying a certificate for entering the round. This brings all honest participants to r within the transmission delay bound. When timeout messages are collected from a quorum of participants, they form a timeout certificate. BDLS may use any of these recommended approaches for round synchronization.

9 The importance of propagating decision messages

During Step 5 of the BDLS protocol, when a participant receives a `decide` message, it propagates the `decide` message to its neighbors. In this section, we show the importance of this process by the potential issues for the HotStuff protocol since it does not have this decision message propagation process.

9.1 HotStuff BFT protocol

HotStuff BFT [22] includes basic HotStuff protocol and chained HotStuff protocol. For simplicity, we only review the basic HotStuff BFT protocol. Similar to PBFT and Tendermint BFT, there are $n = 3t + 1$ participants $P_0, \cdots, P_{n-1}$ and at most t of them are malicious. The `view` is defined and changes in the same way as in PBFT. The major differences between PBFT and HotStuff BFT are:

1. PBFT participants "broadcast" signed messages to all participants though HotStuff participants send the signed messages to the leader participant in a point-to-point channel. In other words, PBFT uses a mesh topology communication network though HotStuff uses a star topology communication network.

2. PBFT uses standard digital signature schemes though HotStuff uses threshold digital signature schemes.

With these two differences, HotStuff achieves authenticator complexity $O(n)$ for both the correct leader scenario and the faulty leader scenario. On the other hand, the corresponding authenticator complexity for PBFT is $O(n^2)$ for the correct leader scenario and $O(n^3)$ for the faulty leader scenario respectively. For simplicity, we will describe the HotStuff BFT protocol using a standard digital signature scheme instead of threshold digital signature schemes. Our analysis does not depend on the underlying signature schemes.

HotStuff BFT has revised the `validRound` and `lockedRound` variables in Tendermint BFT to its `prepareQC` and `lockedQC` variables respectively. Though Tendermint BFT participants set the values for two variables in the same phase, HotStuff BFT participants set the values for these variables in different steps.

In HotStuff BFT, each participant stores a tree of pending commands as its local data structure and keeps the following state variables `viewNumber` (initially 1), `prepareQC` (initially nil, storing the highest QC for which it voted `pre-commit`), and `lockedQC` (initially nil, storing the highest QC for which it voted `commit`).

Each time when a `new-view` starts, each participant should send its `prepareQC` variable to the leader. There is a public function `LEADER(viewNumber)` that determines the current leader participant. When a client sends an operation request m to the leader P_i, the n participants carry out the four phases of the BFT protocol: *prepare, pre-commit, commit* and *decide*.

1. *prepare*: The leader P_i starts the process after it has received $2t+1$ `new-view` messages. Each `new-view` message contains a `prepareQC` variable. P_i selects highQC as the `prepareQC` variable with the highest `viewNumber`. P_i extends the tail of highQC node by creating a new leaf node proposal. P_i then broadcasts the digitally signed new leaf node proposal (together with highQC for safety justification) to all participants in a `prepare` message. A participant accepts this new leaf node proposal if the new node extends the currently locked node *lockedQC.node* or it has a higher `view` number than the current `lockedQC`. If a participant P_j accepts the new leaf node proposal, it sends a `prepare` vote message to P_i by signing it.

2. *pre-commit*: When P_i receives $2t+1$ `prepare` votes for the current proposal, it combines them into a prepareQC. P_i broadcasts prepareQC in a `pre-commit` message. A participant sets its `prepareQC` variable to this received prepareQC value and votes for it by sending the signed prepareQC back to P_i in a `pre-commit` message.

3. *commit*: When P_i receives $2t+1$ `pre-commit` votes. It combines them into a precommitQC and broadcasts it in a `commit` message. A participant sets its `lockedQC` variable to this received precommitQC value and votes for it by sending the signed precommitQC back to P_i in a `commit` message.

4. *decide*: When P_i receives $2t+1$ `commit` votes, it combines them into a commitQC. P_i broadcasts commitQC in a `decide` message. Upon receiving a `decide` message, a participant considers the proposal embodied in the commitQC a committed decision, and executes the commands in the committed branch. The participant increments viewNumber and starts the next `view`.

9.2 What happens if leader does not reliably broadcast `decide` messages in HotStuff

In the following, we describe three scenarios with completely different semantics where the client receives different responses. However, the HotStuff trees are identical for these three scenarios. First assume that at the end of view $v-1$, we have `lockedQC` = `prepareQC` and the HotStuff path corresponding to `lockedQC.node` is $a_0 \to a_1 \to a_l$ where a_0 is the root. Assume that the views v and $v+1$ are executed before GST. That is, the broadcast channel is not reliable before the end of view $v+1$. Assume that the leader for view v is P_i and the leader for view $v+1$ is $P_{i'}$. Furthermore, assume that both P_i and $P_{i'}$ are malicious,

Scenario I: The leader P_i for view v receives $2t+1$ `new-view` messages that contain the identical `highQC` = `prepareQC` with the corresponding path $a_0 \to a_1 \to a_l$. P_i extends the path to the new path $a_0 \to a_1 \to a_l \to b$ and creates a proposal for the new leaf node b. P_i then broadcasts the digitally signed new leaf node proposal (together with `highQC`) to all participants in a `prepare` message. All participant accept this new leaf node proposal and sends a `prepare` vote message to P_i by signing it. In the *pre-commit* phase, P_i receives $2t+1$ `prepare` votes for the current proposal, it combines them into a prepareQC and broadcasts prepareQC in a `pre-commit` message to all participants. All participant set their `prepareQC` variable to this received prepareQC value and vote for it by sending the signed prepareQC back to P_i. During the *commit* phase, P_i receives $2t+1$ `pre-commit` votes. It combines them into a precommitQC and broadcasts it in a `commit` message. All participant set their `lockedQC` variable to this received precommitQC value and vote for it by sending the signed precommitQC back to P_i. In the *decide* phase, P_i receives $2t+1$ `commit` votes, it combines them into a commitQC. P_i only send the commitQC to one honest participant P_j but not to anyone else. After timeout, the view $v+1$ starts. During view $v+1$, the leader participant extends the path $a_0 \to a_1 \to a_l \to b$ to $a_0 \to a_1 \to a_l \to b \to c$ by including a new client command to the node c. Assume that all messages during view $v+1$ are delivered and all participants behaves honestly. Thus at the end of view $v+1$, all participants (except P_j) only executed the commands contained the node c and P_j executed the commands contained both in b and c. Since the client only received one response from P_j that the commands in node b is executed, it will not accept it.

Scenario II: In this scenario, the leader participant P_i for view v does not send any `decide` message in the last step of view v. All other steps are identical to the Scenario I. Thus at the end of view $v+1$, all participants executed the command contained in the node c though no participants executed the command contained in the node b.

Scenario III: In this scenario, the leader participant P_i for view v sends the `decide` message to all participants in the last step of view v. All other steps are identical to the Scenario I. Thus at the end of view $v+1$, all participants executed the commands contained in the nodes b and c.

For all these three scenarios, the path corresponding to the `prepareQC` at the end of view $v+1$ is $a_0 \to a_1 \to a_l \to b \to c$ though the internal states of honest participants are different.

In the HotStuff BFT protocol [22], it is mentioned that "*In practice, a recipient who falls behind can catch up by fetching missing nodes from other replicas*". For all three of the scenarios that we have described, at the end of view $v+1$, the participant who falls behind may fetch the `prepareQC` corresponding to the path $a_0 \to a_1 \to a_l \to b \to c$. But it does not know which scenario has happened. It should be noted that in the HotStuff BFT protocol, the node on the tree only contains the following information: the hash of the parent node and the client command. However, it does not contain any information whether the command has been executed. Our analysis shows that it is important to include in the tree node whether a given command has been executed.

References

[1] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proc. 2nd ACM PODC*, pages 27–30, 1983.

[2] G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *Proc. 3rd ACM PODC*, pages 154–162. ACM, 1984.

[3] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *Preprint arXiv:1807.04938*, 2018.

[4] V. Buterin and V. Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437v4*, 2019.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[6] Cosmos. Cosmos Network: Internet of Blockchains `https://cosmos.network`.

[7] Yvo Desmedt, Yongge Wang, and Mike Burmester. A complete characterization of tolerable adversary structures for secure point-to-point transmissions without feedback. In *International Symposium on Algorithms and Computation*, pages 277–287. Springer, 2005.

[8] D. Dolev and H.R. Strong. Polynomial algorithms for multiple processor agreement. In *Proc. 14th ACM STOC*, pages 401–407. ACM, 1982.

[9] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, 1988.

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

[11] Web3 Foundation. Byzantine finality gadgets, `https://research.web3.foundation/en/latest/polkadot/GRANDPA/`, April 17, 2019.

[12] J. Katz and C.-Y. Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.

[13] J. Kwon. Tendermint powers 40%+ of all proof-of-stake blockchains. invest:asia, available at `https://realsatoshi.net/12886/`, Sept. 12, 2019.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[16] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[17] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[18] TK Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[19] The LibraBFT Team. State machine replication in the Libra Blockchain. available at `https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain/2019-11-08.pdf`, November 28, 2019.

[20] Y. Wang and Y. Desmedt. Secure communication in multicast channels: the answer to Franklin and Wright's question. *Journal of Cryptology*, 14(2):121–135, 2001.

[21] Y. Wang and Y. Desmedt. Perfectly secure message transmission revisited. *Information Theory, IEEE Tran.*, 54(6):2582–2595, 2008.

[22] M. Yin, D. Malkhi, M.K. Reiter, G.G. Gueta, and I. Abraham. HotStuff: BFT consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.