Byzantine Fault Tolerance in Partial Synchronous Networks

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

May 6, 2020

Abstract

The problem of Byzantine Fault Tolerance (BFT) in partial synchronous networks has received a lot of attention in the last 30 years. There are two types of widely accepted definitions for partial synchronous networks. This paper shows that several widely deployed BFT protocols would reach deadlocks in both of these partial synchronous networks (that is, they will not achieve liveness property). To make things worse, it is shown that, for most of the attacks, the adversary only needs to control one participant to carry out the attack instead of controlling $\lfloor \frac{n-1}{3} \rfloor$ participants. Based on the analysis of BFT security requirements for partial synchronous networks, this paper proposes a BFT protocol BDLS and proves its security in partial synchronous networks. It is shown that BDLS is one of the most efficient BFT protocols in partial synchronous networks. Specifically, during synchrony with threshold digital signature schemes, BDLS participants could reach agreement in 3 steps with linear communication/authenticator complexity. It is noted that best existing linear communication/authenticator complexity protocols require at least 7 steps to achieve agreement. The BDLS protocol could be used in several application scenarios such as state machine replication or as blockchain finality gadgets.

Keywords: Byzantine Fault Tolerance; partial synchronous networks; asynchronous networks; communication complexity.

1 Introduction

Lamport, Shostak, and Pease [14] and Pease, Shostak, and Lamport [15] 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.

Fischer, Lynch, and Paterson [10] showed that there is no deterministic protocol for the BFT problem in face of a single failure in an asynchronous networks. 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. To circumvent this impossibility result, 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 [20] 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 [17].

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.

This paper shows that one can launch attacks against several widely deployed BFT protocols (e.g., Tendermint BFT, Ethereum's Casper FFG, 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 even after GST. That is, these BFT protocols could not achieve liveness property in type II partial synchronous networks. For Type I networks, even though all messages are guaranteed to be delivered, one does not know when the message could be delivered. Thus the same attack in the Type II networks can be launched to let the BFT protocol reach deadlock before the end of this unknown message delay time period. On the other hand, all these BFT protocols change views after certain timeout (where the timeout time is much smaller than the unknown message delay time) and after a view change, participants would not accept messages from previous views. That is, even all messages are delivered eventually after an unknown time delay, 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 cannot achieve liveness property in either Type I or Type II partial synchronous 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). It is not our intention to investigate whether Tendermint is secure with its own assumption specified in Tendermint [3] since the network model is not formally defined in [3]. Our result shows that Tendermint is not secure in the commonly accepted Type I and Type II partial synchronous networks. 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 [17] 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 [17]).

Based on the security requirement analysis for BFT protocols in partial synchronous networks, we propose a BFT finality gadget protocol BDLS for blockchains. 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/authenticator complexity. For example, when threshold digital signature schemes are used, HotStuff BFT/LibraBFT achieves linear authenticator complexity with 7 rounds, BDLS achieves linear authenticator complexity with 3 rounds, and the original DLS achieves $O(n^3)$ authenticator complexity. 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 (or even approximate) 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 partial synchronous networks, the protocol designer supplies the consensus protocol first, then the adversary chooses her Δ. For Type II partial synchronous 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, 20, 17] 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 partial synchronous networks for distributed BFT protocol design and analysis. Thus this paper adopts the Type II network model unless specified otherwise.

3 Reliable and strongly reliable broadcast communication primitives

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.

By the above definition, a broadcast channel is unreliable if an honest participant broadcasts a message m to all participants and only a proper subset of honest participants receives this message m. That is, some honest participants receive the message m while other honest participants receive nothing at all (this could happen if the time is before GST in Type II networks). Thus we need to assume that the broadcast channel is unreliable before GST in Type II partial synchronous networks.

The above definition does not say anything about dishonest participants. In practice, a dishonest participant may send different messages to different participants or send the message only to a proper subset of honest participants even after GST in Type II networks. In order to defeat dishonest participants from carrying out these attacks, Bracha [2] designed a *strongly reliable broadcast primitive* with the following additional requirement by assuming that all messages are delivered eventually in the network:

- If a dishonest participant P_i broadcasts a message, then either all honest participants accept the identical message or no honest participant accepts any value from P_i.

Using this strongly reliable broadcast primitive and other validation primitives, Bracha [2] was able to transform Byzantine participants to fail-stop participants in complete asynchronous (or partial synchronous) networks if all message are delivered eventually.

One should take precautions in using such kind of reliable or strongly reliable broadcast primitives for BFT protocol design in partial synchronous networks. For most BFT protocols in partial synchronous networks (e.g., PBFT,

Tendermint BFT, HotStuff BFT, LibraBFT, and our BDLS BFT), there is a timeout setting for each view. If a participant does not receive enough messages for the current view v, it will move to the next view $v+1$ and will no longer accept (that is, discard) messages from a previous view v. Assuming that an honest participant P_i, *using a reliable or a strongly reliable broadcast primitive*, broadcasts a message m during view v which is before GST, some honest participants receive the message m during view v and other honest participants receive the message m during view $v+1$ or later (due to network latency or attacks by dishonest participants). Per the protocol specification, those participants receiving m during $v+1$ or later will discard this message m. That is, even if BFT protocols use a reliable or a strongly reliable broadcast primitive, it is not guaranteed that all honest participants accept the same messages before GST in either Type I or Type II networks. Thus BFT protocols in Type I and Type II partial synchronous networks must have mechanisms to tolerate such kind of scenarios.

One should also take precautions for using reliable or strongly reliable broadcast primitives since these primitives generally have assumptions about the underlying network topology. For example, Bracha's primitive [2] assumes that the underlying network is a complete network. Indeed, it is sufficient to assume that there is a reliable point-to-point communication channel for each pair of participants in Bracha's primitive. 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 [19] and Desmedt-Wang-Burmester [7]). On the other hand, for broadcast communication channels, Wang and Desmedt [18] 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., [14, 19]). In addition to Bracha's strongly reliable broadcast primitive [2], quite a few alternative broadcast primitives have been proposed in the literature using message relays (see, e.g., Srikanth and Toueg [16] and Dwork-Lynch-Stockmeyer [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.

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 most 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: `step` = `propose`, `lockedV` = `nil`, `lockedR` = -1, `validV` = `nil`, and `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 Tendermint BFT cannot achieve liveness in Type II networks

In this section, we show that Tendermint BFT does not achieve the liveness property in Type II partial synchronous networks. Specifically, we show that if a dishonest participant chooses to broadcast a message to a proper subset of participants (this could happen before GST or even after GST), then the system will reach a deadlock and no new block will be created anymore even after the network becomes synchronous. In other words, the Tendermint BFT will reach deadlock before GST and the deadlock could not be removed after GST. In the next section, we extend these attacks against 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 $\mathcal{P}_3 = \{P_{2t+1}, \cdots, P_{3t}\}$. It should be noted that the "broadcast" primitive in [3] is not formally defined. Based on our discussion in Section 3, our following attacks work for any kind of broadcast primitives. That is, no matter whether Tendermint BFT uses regular broadcast primitives, reliable broadcast primitives, or strongly reliable broadcast primitives, the following attacks work against Tendermint BFT.

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.

4.3 Tendermint BFT cannot achieve liveness in Type I networks

Let Δ_T be the round timeout time for the Tendermint BFT protocol. For Type I networks, the adversary chooses the network delay time Δ based on the given protocol and Δ_T. Assume that the adversary chooses $\Delta = 10\Delta_T$. The

value of Δ is unknown to the honest participants. That is, if a participant P_i (honest or dishonest) sends a message m to a participant P_j at time t, P_j receives this message m during the period $[t, t+\Delta]$. In the following, we revise the **Attack 1** in Section 4.2 so that it works against Tendermint BFT in Type I networks with time delay Δ.

Attack 1r. 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 all participants with the following properties

- Participants in $\mathcal{P}_1 \cup \mathcal{P}_2$ receive this PROPOSAL as soon as possible (e.g., within time delay Δ_T).
- Participants in $\mathcal{P}_3$ receive this PROPOSAL with a time delay of at least $9\Delta_T < \Delta$.

After receiving $\langle \text{PROPOSAL}, h, 0, v, -1\rangle$ from P_0, each participant $P_j \in \mathcal{P}_1$ broadcasts $\langle \text{PREVOTE}, h, 0, H(v)\rangle$ with the following properties

- Participants in $\mathcal{P}_2$ receive this PREVOTE as soon as possible (e.g., within time delay Δ_T).
- Participants in $\mathcal{P}_1 \cup \mathcal{P}_3$ receive this PREVOTE with a time delay of at least $9\Delta_T < \Delta$.

After receiving P_0's message $\langle \text{PROPOSAL}, h, 0, v, -1\rangle$, each participant $P_j \in \mathcal{P}_2$ sets $\texttt{step} = \texttt{prevote}$ and broadcasts $\langle \text{PREVOTE}, h, 0, H(v)\rangle$ to all participants as soon as possible. Each participant $P_j \in \mathcal{P}_2$ receives $2t+1$ messages $\langle \text{PREVOTE}, h, 0, H(v)\rangle$ within timeout Δ_T. Thus 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 $\langle \text{PRECOMMIT}, h, 0, H(v)\rangle$ as soon as possible. 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 Δ_T 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 Δ_T 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. Thus the process will continue forever without making an agreement for the height h even after GST. It should be noted that at some round (e.g., after round 8), participants in $\mathcal{P}_3$ receive P_0's PROPOSAL message for round 0 and participants in $\mathcal{P}_1 \cup \mathcal{P}_3$ receive the PREVOTE message for round 0. However, since all participants are not in round 0, they will discard these PROPOSAL and PREVOTE messages. Though it is not explicitly mentioned in [3] that a participant should discard the message from a previous round, this follows from the description of "Algorithm 1 Tendermint consensus algorithm" in [3]. That is, in the line 28 and line 44, when a participant receives a a PROPOSAL or PREVOTE message, it will check whether a message round matches the current participant round, it will only take action if the round matches. Otherwise, it takes no action (that is, discard that message). Indeed, if a participant is allowed to process/accept messages from a previous round, then the lock-mechanisms will make no sense and the protocol will not be safe. It is further noted that in PBFT, it is explicitly mentioned that after a participant changes its view, it will discard any future received messages from a previous view.

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+Δ 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 achieve linear 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. 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 $(\langle h, r\rangle_j, \langle h, r, B'_j\rangle_j)$ to the leader P_i where $B'_j \in$ BLOCK$_j$ is the maximal acceptable candidate block for P_j. The message $\langle h, r\rangle_j$ is considered as a round-change message. After sending the round-change message, P_j will not accept messages except a "decide" message for round $r' < r$ anymore.

2. If P_i receives at least $2t + 1$ round-change messages (including himself), it enters round r (see Section 9 for details on when P_i can stop waiting for more round-change request messages). In these round-change messages, if there are at least $2t+1$ signed messages from $2t+1$ participants with the same candidate block $B' \neq NULL$, 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$_i$ and broadcasts $\langle \text{select}, h, r, B'', \text{proof}\rangle$ 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. It should be noted that in order to achieve linear communication complexity when a threshold signature scheme employed, the "proof" in the `lock`-message and `select`-message are different: In the `lock`-message, the "proof" contains an assembled digital signature on the message $\langle h, r, B'\rangle$ while, in the `select`-message, the "proof" contains an assembled digital signature on the message $\langle h, r\rangle$. See Remark 3 for details.

3. If a participant P_j (including P_i) receives a valid $\langle \text{select}, h, r, B'', \text{proof}\rangle$ from P_i during Step 2, then it adds B'' to its BLOCK$_j$. If a participant P_j (including P_i) 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 (including P_i) 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 2 for more details on this). Otherwise, it goes to the following lock-release step:

   - (*lock-release*) If a participant P_j (including P_i) 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*: At any time during the protocol, 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*: At any time during the protocol, if P_j receives a valid "lock" or "select" or "decide" message for a round $r' > r$, P_j moves to round r' and processes the "lock" or "select" or "decide" message.

8. *timeout*: For each step, P_j should set an appropriate timeout counter. If P_j does not receive enough messages to move forward before timeout counter expires, it moves to the next step. The reader is referred to Section 8 and Section 9 for a detailed discussion of round/height synchronization.

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.

Remark 3: In order to achieve linear communication/authenticator complexity with threshold digital signature schemes, participant P_j sends the signed message $(\langle h, r \rangle_j, \langle h, r, B'_j \rangle_j)$ to the leader P_i during step 1. It should be noted that if there are $2t+1$ participants that send the same B'_j to the leader, then the leader P_i can assembly a signature for $\langle h, r, B'_j \rangle$. If there is no such value B'_j, then the leader can only assembly a digital signature for $\langle h, r \rangle$ which can be used for the `select` message. In the security proof for BDLS in the next section, the leader does not need to assemble a digital signature for B'_j if it only broadcasts a `select` message.

7.2 Liveness and Safety

The security of BDLS protocol is proved by establishing a series of Lemmas. The proofs for Lemmas 1, 2, 3 and Theorem 1 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 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''. □

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 [20]	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$

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 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 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 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., [20]). 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 status 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. The details are presented in Section 9.

9 BDLS with Pacemaker

Though BDLS may use the PBFT mechanism to keep round synchronization (that is, the timeout period for round r is $2^r\Delta$), it seems to be more efficient to use Pacemaker for BDLS round synchronization. Similar to LibraBFT, the advancement of rounds in BDLS is governed by a module called Pacemaker. The Pacemaker keeps track of votes and of time. We revise BDLS slightly so that a Pacemaker could be seamlessly integrated into the protocol without extra workload. The major change is Step 1 where Pacemaker timeout messages are combined with round-change requests for efficiency. The round r of the height h for a participant P_j starts when its Pacemaker receives round-change messages from at least $2t+1$ participants or if its timeout for round $r-1$ or if it receives a "lock" or a "select" or a "decide" message for round r. Specifically, the round r proceeds as follows where $P_i = leader(h, r)$ is the leader for round r:

1. (If $r > 0$, this is done at the end of round $r-1$ of height h. If $r = 0$, this is done after a decision for height $h-1$ is made) The Pacemaker of each participant P_j (including P_i) broadcasts the signed message $(\langle h, r\rangle_j, \langle h, r, B'_j\rangle_j)$ where $B'_j \in \text{BLOCK}_j$ is the maximal acceptable candidate block for P_j of height h. The message $\langle h, r\rangle_j$ is considered as a round-change message for round r. After P_j broadcasts the round-change message for round r, it will set a timeout message Δ_0 and enters `roundchanging` status. During `roundchanging` status, a participant will not accept any messages except round-change messages and "decide" messages for the height h of any round. Furthermore, if $r > 0$, then each participant P_j (including P_i) initializes all of its variables except the locked block variable. If $r = 0$, then each participant P_j (including P_i) initializes all of its variables including the locked block variable. For any participant P_j who is in `roundchanging` status, if it does not enter the `lock` status of Step 2 before Δ_0 expires, it resends the round-change message and resets its Δ_0.

2. During any time of the protocol, if the Pacemaker of P_j (including P_i) receives at least $2t+1$ round-change messages (including round-change message from himself) for round r (which is larger than its current round status), it enters `lock` status of round r. If P_j has not broadcast the round-change message yet, it broadcasts now. Then P_j sets the timeout counter Δ_1 for `lock` status[1]. Furthermore, as soon as the leader P_i enters `lock` status of round r, it starts a timeout counter $\Delta'_1 < \Delta_1$ concurrently[2]. The leader P_i stops the time counter Δ'_1 as soon as he receives n round-change requests or as soon as he receives $2t+1$ round-change requests with an identical proposed block. As soon as the time counter Δ'_1 expires or the leader P_i stops the time counter Δ'_1, P_i distinguishes the following two cases:

   (a) Among all round-change messages that P_i has received, if there are at least $2t+1$ signed messages from $2t+1$ participants with the same candidate block $B' \neq NULL$, then P_i broadcasts the following signed message (2) to all participants

   $$\langle \text{lock}, h, r, B', \text{proof}\rangle_i \tag{5}$$

   where proof shows that at least $2t+1$ participants singed B' (the proof also shows that round-change request has been authorized by at least $2t+1$ participants).

   (b) If P_i does not receive such a block B', then P_i adds all received candidate blocks to its local variable BLOCK_i and broadcasts

   $$\langle \text{select}, h, r, B'', \text{proof}\rangle \tag{6}$$

   where B'' is the candidate block $B'' = \max\{B : B \in \text{BLOCK}_i\}$ and proof shows that round-change request has been authorized by at least $2t+1$ participants from Step 1.

[1] The `lock` status timeout counters could be set as follows: For round $r = 0$, the timeout counter $\Delta_1 = \Delta_{1,0}$ should be at least 4 network transmission delays plus some time for each participant to process the messages. For round $r > 0$, the timeout counter could be defined as $r\Delta_{1,0}$.

[2] Though it is sufficient for a non-leader participant to collect only $2t+1$ round-change requests, the leader should collect as many round-change message as possible. In particular, the leader should try to collect all round-change messages from all participants. It is recommended that after the leader P_i collects $2t+1$ round-change requests and starts the `lock` status timeout counter Δ_1, it initiates another timeout counter $\Delta'_1 < \Delta$ to collect as many as possible round-change requests if more round-change requests still arrive. Generally, we can set Δ_1 as two network transmission delays. This mechanism is used to avoid the following attack: the malicious t participants may send random round-change messages to the leader. If the leader only checks the first $2t+1$ messages (among them, t could be malicious), then the system may never reach an agreement. However, the leader should not wait forever since the t malicious participants may choose not to send round-change request at all.

3. If a participant P_j (including P_i) does not receive a valid message from the leader P_i during Step 2 and the timeout counter Δ_1 expires, P_j enters `commit` status of round r and sets the timeout counter Δ_2 for `commit` status[3]. Otherwise, if a participant P_j (including P_i) receives a valid message (5) or (6) from P_i before Δ_1 expires, P_j stops the time counter Δ_1 and distinguishes the following two cases:

   - If P_j receives a valid $\langle \text{select}, h, r, B'', \text{proof} \rangle$ from P_i during Step 2, then it adds B'' to its BLOCK$_j$ and enters `lock-release` status of round r and sets the timeout counter Δ_3 for `lock-release` status.
   - If P_j (including P_i) 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 and enters `commit` status by setting the timeout counter Δ_2:
     (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 (5)
     (c) sends the following signed commit message to the leader P_i.

$$\langle \text{commit}, h, r, B' \rangle_j. \tag{7}$$

4. If P_i receives at least $2t + 1$ commit messages (7) for the round r of height h with the locked value B' of (5) before Δ_2 expires, then P_i *decides* on the value B' and broadcasts the following decide message to all participants

$$\langle \text{decide}, h, r, B', \text{proof} \rangle_i. \tag{8}$$

   where proof is a list of at least $2t + 1$ commit messages (7).

5. If a participant P_j (including P_i) receives a decide message (8) from Step 4 or from its neighbor before the timeout counter Δ_2 expires, it decides on the block B' for B^h and the Pacemaker of P_j goes to Step 1 of height $h+1$. At the same time, the participant P_j propagates (broadcasts) the decide message (8) to all of its neighbors if it has not done so yet. Otherwise, if P_j (including P_i) does not receive a `decide` message from the leader P_i or its neighbors before the timeout counter Δ_2 expires, P_j enters `lock-release` status of round r and sets the timeout counter Δ_3 for `lock-release` status[4].

6. (*lock-release*) If a participant P_j (including P_i) has some locked values, then P_j calculates

$$r_1 = \max\{r' : P_j \text{ holds a lock } \langle \text{lock}, h, r', B', \text{proof} \rangle_{i'}\}.$$

   P_j releases all locks $\langle \text{lock}, h, r'', B'', \text{proof} \rangle_{i''}$ with $r'' \neq r_1$. P_j then broadcasts the following lock-release message

$$\langle \text{lock-release}, h, r, \langle \text{lock}, h, r_1, B', \text{proof} \rangle_{i_1} \rangle. \tag{9}$$

   If P_j receives a lock-release message $\langle \text{lock-release}, h, r, \langle \text{lock}, h, r'_1, B''', \text{proof} \rangle_{i'_1} \rangle$ with $r'_1 > r_1$ from another participant before the timeout Δ_3 expires, then P_j releases its lock $\langle \text{lock}, h, r_1, B', \text{proof} \rangle_{i_1}$ and records the lock $\langle \text{lock}, h, r'_1, B''', \text{proof} \rangle_{i'_1}$. After the timeout Δ_3 expires, the Pacemaker of P_j goes to Step 1 for round $r + 1$ of height h.

7. *height synchronization*: At any time of the protocol run, if P_j receives a finalized bock of height h (e.g., a decide message (8)), P_j decides for height h and moves to height $h + 1$.

8. *round synchronization*: At any time of the protocol run, if P_j receives a valid "lock" or "select" or "decide" message for a round $r' > r$, P_j moves to round r' and process the "lock" or "select" or "decide" message. Furthermore, at any time, if P_j receives from more than $t + 1$ participants valid messages for round $r' > r$ (including round-change messages for round r'), P_j goes to Step 1 for round r' of height h.

[3]The `commit` status timeout counters could be set as follows: For round $r = 0$, the timeout counter $\Delta_2 = \Delta_{2,0}$ should be at least 2 network transmission delays plus some time for each participant to process the messages. For round $r > 0$, the timeout counter could be defined as $r\Delta_{2,0}$.

[4]The `lock-release` status timeout counters could be set as follows: For round $r = 0$, the timeout counter $\Delta_3 = \Delta_{3,0}$ should be at least 2 network transmission delays plus some time for each participant to process the messages. For round $r > 0$, the timeout counter could be defined as $r\Delta_{3,0}$.

10 Static and dynamic BFT participants

For blockchain environments, the BFT participants may changes from height to height (or even from round to round). In order to obtain the BFT participant team, each participant should use an API call to obtain the participant list for the height h before submitting the round-change message for a new height h. However, for a permissionless blockchain, the full participant list may not be available at the time when it submits the round-change message. Thus each time, when a participant receives a BFT message, it should check whether the sender of the message is in its local list of participants or not. If not, it should use an API to check whether the sender is a qualified participant for this height or not. If it is a qualified participant, it should expand its participant list and adjust the parameters accordingly.

On the other hand, some applications of BDLS BFT protocol have static BFT participants. To make the BDLS package more efficient for these applications, one should use an API call to check whether BFT participants change from round to round. If the participant list does not change, the BLDS protocol should not carry out the extra checks discussed in the preceding paragraph.

11 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.

11.1 HotStuff BFT protocol

HotStuff BFT [20] 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`.

11.2 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 $a_0 \rightarrow a_1 \rightarrow a_l$ 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 $a_0 \rightarrow a_1 \rightarrow a_l$. P_i extends the path to the new path $a_0 \rightarrow a_1 \rightarrow a_l \rightarrow b$ 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 $a_0 \rightarrow a_1 \rightarrow a_l \rightarrow b$ to $a_0 \rightarrow a_1 \rightarrow a_l \rightarrow b \rightarrow c$ 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 $a_0 \rightarrow a_1 \rightarrow a_l \rightarrow b \rightarrow c$ though the internal states of honest participants are different.

In the HotStuff BFT protocol [20], 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 $a_0 \rightarrow a_1 \rightarrow a_l \rightarrow b \rightarrow c$. 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. *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, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

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

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

[17] 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.

[18] 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.

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

[20] 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.