Another Look at Byzantine Fault Tolerance

Yongge Wang
UNC Charlotte

July 24, 2019

Abstract

We review several solutions for the Byzantine Fault Tolerance (BFT) problem and discuss some aspects that are frequently overlooked by existing literatures. For example, PBFT and HotStuff BFT protocols (HotStuff has been adopted by Facebook Libra) require a reliable broadcast primitive. We show that if the broadcast primitive is not reliable then the PBFT and HotStuff BFT protocols could not achieve the liveness property (that is, the system will never reach an agreement on a proposal). Though these BFT protocols have been developed for partial synchronous networks, we show that they cannot achieve consensus in partial synchronous networks since the participants do not know what is the Global Stabilization Time (GST) and broadcast channels before GST are defined to be unreliable (e.g., DoS attacks on certain participants). Thus it is important for developers to be aware of these issues when developing applications (such as blockchains) using these BFT protocols.

1 Introduction

Lamport, Shostak, and Pease [12] and Pease, Shostak, and Lamport [14] initiated the study of reaching consensus in face of Byzantine failures and designed the first synchronous solution for Byzantine agreement. Dolev and Strong [7] 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 [11] proposed an expected constant-round BFT protocol in a synchronous network setting againt $\lfloor \frac{n-1}{2} \rfloor$ Byzantine faults.

For an asynchronous network, Fischer, Lynch, and Paterson [9] 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 have two assumptions: (1) when a process writes a bit on the output register, it is finalized and can not change anymore; and (2) a nonfaulty process runs infinitely many steps in a run. If either of these assumptions is removed, then the construction will not work and the result may not hold any more. As noted in Andreina et al [1] "*While it is impossible to provably achieve safety and liveness in fully asynchronous networks [9], blockchain protocols target eventual consistency, resp liveness, meaning that the chain of some honest party may be inconsistent, resp. stuck, during asynchronous periods but then it will catch up as soon as* *the network obeys a synchronous regime.*" With this relaxed requirement, the first assumption in Fischer et al's result could be removed in blockchain consensus protocol design. By removing the first required assumption, we will be able to construct consensus protocols to tolerate $\lfloor \frac{n-1}{3} \rfloor$ Byzantine faults in fully asynchronous networks. In certain sense, our construction is similar to the finite injury priority method (Muchnick [13] and Friedberg [10]) that was used to construct the intermediate degrees between $\mathbf{0}$ (the minimum Turing-degree) and $\mathbf{0}'$ (the maximum recursively enumerate degree) in computability theory.

Several researchers have tried to design BFT consensus protocols to circumvent the impossibility. For example, Ben-Or [3] initiated the probabilistic approach to BFT consensus protocols in completely asynchronous networks and Dwork, Lynch, and Stockmeyer [8] designed BFT consensus protocols in partial synchronous networks. Castro and Liskov [6] developed a practical BFT consensus protocol PBFT in partial synchronous networks. PBFT has been deployed in several practical systems. Recently, Yin et al [19] improved the PBFT protocol by changing the mesh communication network in PBFT to star communication networks in HotStuff and by using threshold cryptography. Facebook blockchain has adopted HotStuff in their LibraBFT protocol.

Both PBFT and HotStuff BFT protocols have used reliable "multicast" or "broadcast" protocols for certain message transmission. This assumption is very strong and may be infeasible in practice. Katz and Koo [11] mentioned that "*Byzantine agreement readily implies protocols for broadcast*". On the other hand, it may be much easier to design BFT consensus protocols based on the existence of reliable broadcast channels (see, e.g., Wang and Desmedt [16], Lamport et al [12] and Buterin [5]). We show that if there does not exist a reliable broadcast channel, we can launch an attack on PBFT and an attack on HotStuff so that participants would never reach an agreement on a proposal. Furthermore, PBFT and HotStuff BFT are designed for partial synchronous networks. By the definition of partial synchronous networks in [6, 19], the network is not reliable before the time GST and the participants do not know what is the value of GST. Before the time GST, the package could get lost and DoS attacks may be launched. Thus the broadcast channels are not reliable before GST. Thus our attack also shows that PBFT and HotStuff BFT cannot achieve consensus in partial synchronous networks. This contradicts the claims in [6, 19].

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 [8] considered the following two kinds of partial synchronous networks:

- $\Delta < \infty$ is unknown. That is, there exisits a Δ but the participants do not know the exact value of Δ.
- $\Delta < \infty$ 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 the first type of partial synchronous networks, the protocol designer supplies the consensus protocol first, then the adversary chooses her Δ. For the second type of partial synchronous networks, the adversary picks the Δ and the protocol designer (knowning Δ) supplies the consensus protocol, then the adversary chooses the GST T. The definition of partial synchronous networks in [6, 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.

3 PBFT and HotStuff BFT

3.1 PBFT

We first briefly review the practical BFT protocol PBFT. In PBFT, there are $n = 3t+1$ participants $P_0, \cdots, P_{n-1}$ and at most t of them are malicious. The PBFT protocol proceeds through a succession of configurations called `views`. In each `view` period, one participant is called the *leader* (or the *primary*) and others are called *backups*. It is assumed that the numbering of participants are known to all. For the `view` number v ($v = 0, 1, 2, \cdots$), the leader participant is P_i where $i \equiv v \mod n$. The `view` period changes when it appears that the leader participant has failed.

Each time when a client sends an operation request to the leader participant, the n participants carry out the three phases of the PBFT protocol: *pre-prepare, prepare,* and *commit*. The pre-prepare and prepare phases are used to order requests in the same view and the prepare and commit phases are used to ensure that committed requests are totally ordered across views. For an operation request m during `view` period v, let P_i be the leader (that is, $i \equiv v \mod n$). The protocol proceeds as follows:

1. *pre-prepare*: The leaderP_i **multicasts** the signed message

   $$sig_i(\langle \text{PRE-PREPARE}, v, seq, H(m)\rangle) \tag{1}$$

   to all participants, where seq is a unique sequence number. A participant P_j accepts the pre-prepare message (1) if

   - the signarure is valid and the current view number is v;

- it has not accepted another pre-prepare message with identical (v, seq) but different hash value $H(m)$.
- the sequence number seq is within the requied interval.

2. *prepare*: If the participant P_j accepts the pre-prepare message (1), it enters the *prepare* phase by **multicasting** a digitally signed message

$$sig_j(\langle \text{PREPARE}, v, seq, H(m), P_j \rangle) \tag{2}$$

   to all participants and inserting both messages (1) and (2) to its log.

3. *commit*: The tuple $\langle m, v, seq, P_j \rangle$ is called prepared for the participant P_j if and only if P_j has inserted the message (1) into its log and has collected $2t$ valid signatures on $\langle \text{PREPARE}, v, seq, H(m), P_{j'} \rangle$ for $j \neq j'$. When $\langle m, v, seq, P_j \rangle$ is prepared for participant P_j, P_j **multicasts** the digitally signed message

$$sign_j(\langle \text{COMMIT}, v, seq, H(m), P_j \rangle) \tag{3}$$

   to all participants. Participants accept commit messages and insert them in their log provided they are properly signed and valid. A tuple $\langle m, v, seq, P_j \rangle$ is *local-committed* if and only if the tuple $\langle m, v, seq, P_j \rangle$ is prepared for P_j and P_j has accepted $2t + 1$ commits (including its own) from different participants. After each participant P_j local-commis a tuple $\langle m, v, seq, P_j \rangle$, it will execute the operations contained in the message m.

In PBFT, a tuple $\langle m, v, seq \rangle$ is *committed* if and only if the tuple $\langle m, v, seq, P_j \rangle$ is prepared for $t + 1$ honest participants P_j. Castro and Liskov [6] showed that: If a tuple $\langle m, v, seq, P_i \rangle$ is local-committed for some non-faulty participant P_i then the tuple $\langle m, v, seq \rangle$ is committed.

It is mentioned in [6] that if some participant misses some messages that were discarded by all honest participants, it will need to be brought up to date by transferring all or a portion of the service state. For this kind of transfer service, the participant needs some proof that the state is correct. To improve performance, it is recommended in [6] to produce checkpoints for sequence numbers divisible by some constant (e.g., 100). When a participant P_j produces a checkpoint, it multicasts a message

$$sig_j(\langle \text{CHECKPOINT}, seq, H(state), P_j \rangle) \tag{4}$$

to all participants. In order to prove the correctness for one checkpoint, a participant needs to collect at least $2t + 1$ checkpoint signatures (4) from $2t + 1$ participants.

3.2 HotStuff BFT

HotStuff BFT [19] includes basic HotStuff protocol and chained HotStuff protocol. For simplicity, we only review the basic HotStuff BFT protocol. Our analysis works for chained HotStuff also. Similar to PBFT, 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 "multicast" 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 signatue scheme instead of threshold digital signature schemes. Our analysis does not depend on the underlying signature schemes.

In HotStuff BFT, each participant stores a tree of pending commands as its local data structure. Each time when a `new view` starts, each participant should send its highest prepareQC branch of its local tree to the leader participant. During the `view` period v, P_i serves as the leader participant if $i \equiv v \mod n$. 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 selects the branch highQC that has the highest preceding view among all the `new view` messages (in which a prepareQC was formed) it received from $2t+1$ participants. 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 message if the signature is valid and if

   - the branch of this new leaf node extends from the currently locked node *lockedQC.node* or
   - the new node has a higher `view` number than the current locked QC.

   If a participant P_j accepts the new leaf node proposal message, 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 responds to P_i with `pre-commit` vote by signing it.

3. *commit*: When P_i receives $2t + 1$ `pre-commit` votes, it combines them into a precommitQC and **broadcasts** it in a `commit` message; participants respond with a `commit` vote and set its lockedQC to precommitQC.

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`.

4 Multicast communication channels

The difference between point-to-point communication channels and multicast communication channels has been extensively studied in the literature. 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. It is well known that $(2t + 1)$-connectivity is necessary for reliable communication againt t Byzantine faults (see, e.g., Wang and Desmedt [17]). On the other hand, for multicast communication channels, Wang and Desmedt [16] 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 multicast 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 multicast channel, it will be observed by all receivers.

Though multicast channels at physical layers are commonly used in local area networks, it is not trivial to design reliable multicast channels over the Internet infrastructure. The situation is more complicated due to the fact that the Internet connectivity is not a complete graph and some direct communication paths between participants are missing (see, e.g., [12, 17]). In the literature, there are mainly two approaches to obtain multicast communication channels. In the first approach, multicast communication primitives are constructed using Byzantine agreement protocols (see, e.g., Katz and Koo [11]). In other words, one first needs to design a BFT protocol to obtain a secure multicast protocol. This is commonly used in cryptographic multi-party computation protocols. The second approach is based on message relays (see, e.g., Srikanth and Toueg [15], Dwork, Lynch, and Stockmeyer [8], and LibraBFT [2]). In the message relay based multicast protocol, if an honest participant accepts a message signed by another participant, it relays the signed message to other participants. The following properties are required for the multicast channels in [15, 8]:

1. (Correctness) If an honst participant broadcasts a signed message m in round k, then every honest participant accepts m in the same round.

2. (Unforgeability) If a participant is honest and does not broadcast a message m, then no honest participant ever accepts m.

3. (Relay) If an honest participant accepts a signed message m in round k, then every other honest participant accepts m in round $k + 1$.

The LibraBFT [2] claims the following property for its multicast channel:

1. *(probabilistic-reliable-broadcast) After GST, if an honest node receives or possesses data that requires gossiping, then—with high probability—before an (unknown) time delay, every other honest node will have received these data.*

Though these properties may be true after GST, it is not true before GST since messages before GST could get lost or re-ordered by the definition. On the other hand, it is even not clear whether it is realistic to assume these multicast properties after GST in the Internet infrastructure since the t malicious participants may re-write the multicast primitive in the client code or mount DoS attacks at any time. Of course, it seems that the models in PBFT [6], HotStuff BFT [19], and LibraBFT [2] do not allow DoS attacks after GST.

A multicast channel is said to be *unreliable* if a malicious participant could multicast a message m_1 to a proper subset of the participants but not to all participants. That is, some participant will receive the message m_1 while other participants will receive a different message m_2 or receive nothing. The BFT protocols in PBFT [6], HotStuff BFT [19], and LibraBFT [2] assume the existence of a reliable multicast channel after GST. We will show in Section 4.2 that, if the multicast channel is unreliable (either before GST or after GST), then these BFT protocols are insecure. On the other hand, our attacks Section 4.2 do not work against the DLS BFT protocol in Dwork, Lynch, and Stockmeyer [8]. The reader is referred to Appendix for a brief discussion of the DLS BFT protocol.

4.1 What happens if there exists reliable multicast channels?

As we have mentioned in the preceding paragraphs, in order for PBFT [6], HotStuff BFT [19], and LibraBFT [2] to be secure, one needs to have a reliable multicast channel before and after GST. By the definition of GST, multicast channels before GST are obviously unreliable. But let us just assume that we have "reliable multicast channels" before GST. We show that if a reliable multicast channel exists before and after GST, then there exist more efficient and robust BFT protocols. For example, in the simple BFT protocol, only one message flow is required instead of three (or four) messge flows in PBFT/HotStuff and the simple BFT protocol is secure against $n-1$ or $\lfloor\frac{n-1}{2}\rfloor$ Byzantine faults instead of $\lfloor\frac{n-1}{3}\rfloor$ Byzantine faults in PBFT/HotStuff.

Indeed, Lamport et al [12] has already considered these simple BFT protocols in reliable multicast channels. Assuming that "if one honest node has seen a particular value (validly), then every other honest node has also seen that value", the BFT solution with signed messages by Lamport et al [12] achieves consensus with one message flow. The reader is also referred to Buterin [5] for a more detailed explanation of the protocol and its applications in blockchains.

Assume that the multicast channel is reliable and there is a strict total order on all messages. That is, for any two different messages $m_1 \neq m_2$, either $m_1 < m_2$ or $m_2 < m_1$. Then the following protocol achieves consensus againt $n-1$ Byzantine faults. For a view period that P_i is the leader participant, the protocol proceeds as follows

- The leader P_i multicasts the message m to all participants.
- We distinguish the following three cases:
  - If a participant receives no valid message, it accepts a default value.
  - If a participant receives only one valid message, it accepts the message and executes the commands within it.

– If a participant receives multiple valid messages $m_1, \cdots, m_k$, it accepts the minimal valid message m_i such that $m_i \leq m_j$ ($j = 1, \cdots, k$) and executes the commands within it.

Now we show that the above simple protocol is a secure BFT protocol. If P_i is honest, then all honest participants receive the same message and excute the same command. If P_i is dishonest, it may multicast several messages. However, P_i cannot multicast different messages to different participants. That is, all honest participants receive the same set of valid messages. Thus all honest participants accept the same valid message and execute the same command. It is straightforward to show that, for State Machine Replication (SMR) scenario, the above protocol can tolerate $\lfloor \frac{n-1}{2} \rfloor$ Byzantine faults.

4.2 What happens if multicast channel is unreliable before GST?

In the LibraBFT, it has the following statement "**Asynchrony**: *Consistency is guaranteed even in cases of network asynchrony (i.e., during periods of unbounded communication delays or network disruptions). This reflects our belief that building internet-scale consensus protocol whose safety relies on synchrony would be inherently both complex and vulnerable to Denial-of-Service (DoS) attacks on the network".* In other words, the multicast channel is unreliable before GST. In the following, we show that if the multicast channel is unreliable (even before GST), neither PBFT nor HotStuff BFT is secure. For both protocols, we consider one specific *view* period where 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 $\mathcal{P}_3 = \{P_{2t+1}, \cdots, P_{3t}\}$.

An attack on PBFT: In the *pre-prepare* step, the malicious P_0 multicasts the PRE-PREPARE message (1) to participants in $\mathcal{P}_1$ and $\mathcal{P}_2$ (but not to participants in $\mathcal{P}_3$). During the *prepare* step, participants in $\mathcal{P}_1$ multicast the PREPARE message (2) to participants in $\mathcal{P}_1$ and $\mathcal{P}_2$ but not to participants in $\mathcal{P}_3$. However, participants in $\mathcal{P}_2$ multicast the PREPARE message (2) to all participants. Now the tuple $\langle m, v, seq, P_j \rangle$ is prepared for participants P_j in $\mathcal{P}_1$ and $\mathcal{P}_2$ (but not prepared for participants in $\mathcal{P}_3$). During the *commit* step, participants in $\mathcal{P}_1$ multicast the COMMIT message (3) to participants in $\mathcal{P}_1$ and $\mathcal{P}_2$. Participants in $\mathcal{P}_2$ multicast the COMMIT message (3) to all participants. At the end of the *view* period v, the tuple $\langle m, v, seq, P_i \rangle$ is *local-committed* for participants in $\mathcal{P}_1$ and $\mathcal{P}_2$ (but not for participants in $\mathcal{P}_3$). That is, all participants in $\mathcal{P}_1$ and $\mathcal{P}_2$ execute the command contained in m though no participant in $\mathcal{P}_3$ executes the command. If participants in $\mathcal{P}_1$ would not multicast checkpoint signatues, participants in $\mathcal{P}_3$ could not prove the correctness of the checkpoints published by participants in $\mathcal{P}_2$. Thus honest participants in $\mathcal{P}_3$ will have different states and could not get updated to the states for participants in $\mathcal{P}_2$. Furthermore, if participants in $\mathcal{P}_1$ would "behaves as honest participants" from now on (but not multicast checkpoint signatures), then participants in $\mathcal{P}_3$ would "look like dishonest" participants. In a summary, if the above attack is launched before GST, the protocol would not be able to achieve consensus for honest participants even after GST.

The above attack on the PBFT protocol may be resolved by only requiring $t + 1$ signatures on the checkpoint messages instead of the currently required $2t + 1$ signatures and by requiring more frequent publications of checkpoint messages. But these changes will increase the number of message flows in the system significantly. It should be noted that the adversary may mount DoS attacks on the checkpoint message also. Furthermore, there may exist other potential attacks on the PBFT protocol. Alternative, one may fix PBFT in the following way: when an honest participant P commit a proposal, it should immediately propagate it to other participants. Each participant should accept this committed proposal as long as it contains $t+1$ signatures. Eventually all honest participants will accept this committed proposal if the participant is not disconnected from P. By disconnection from P, we mean that all message paths from the participant to P contains a malicious participants. This raises another important question that one should analyze when designing BFT protocols: the connectivity of the participant networks (see, e.g., [12, 17]).

An attack on HotStuff BFT: This attack is similar to the attack on the PBFT. That is, the adversary forces some honest participants (but not all honest participants) to accept a message. During the *prepare* step, the leader participant P_0 (who is malicious) multicasts the *prepare* message to participants in $\mathcal{P}_1$ and $\mathcal{P}_2$ but not to participants in $\mathcal{P}_3$. Durng the *pre-commit* step, P_0 multicasts the *pre-commit* message to participants in $\mathcal{P}_1$ and $\mathcal{P}_2$ but not to participants in $\mathcal{P}_3$. During the *commit* step, P_0 multicasts the *commit* message to 50% of participants in $\mathcal{P}_2$ (but neither to participants in $\mathcal{P}_3$ nor to the other 50% of participants in $\mathcal{P}_2$). Now 50% of participants $\mathcal{P}_2$ (but neither participants in $\mathcal{P}_3$ nor the other 50% of participants in $\mathcal{P}_2$) set its lockedQC to precommitQC during this step. P_0 will not post any message after this step. An asynchronous network scheduling causes participants to move to *view* $v+1$ without receiving *decide* message. The malicious participant P_1 is the leader for the *view* period $v+1$. During *view* $v+1$, P_1 uses the same approach to force some other honest participants to set its lockedQC to another precommitQC whose node is conflict with the precommitQC node in *view* period v. Now a similar argument as in the infinite non-deciding scenario for a "two-phase" HotStuff (see section **Livelessness with two-phases** of [19]) could be used to show that the HotStuff BFT will never reach on any consensus in future.

In the above attack on HotStuff BFT, the adversary manipulated the broadcast channel so that some honest participants lock conflict branches of the trees. Using the same approach, a malicious leader could force $t+1$ (or less or more) honest participants to accept a *decide* message and executes the commands in the commited branch. But other honest participants will do nothings and move to the next *view* without executing any command.

Discussion. In this section, we showed that if reliable multicast communication channel does not exist and if a malicious participant could select the subset of participants as the receiver of the multicast channel, then PBFT and HotStuff BFT (and several other BFT protocols in the literature) are insecure. On the other hand, if we assume that there is a reliable multicast channel, then we showed in Section 4.1 that we do not need complicated protocols such as PBFT/HotStuff to achieve BFT consensus.

5 BFT in asynchronous networks with eventual consistency

Fischer, Lynch, and Paterson [9]'s impossibility results for deterministic BFT protocols in face of a single failure has the following assumptions. Each participant has an output register with values in $\{0, 1, b\}$. The output register starts with value b. The states in which the output register has value 0 or 1 are distinguished as being decision states. Once it reaches a decision state, one can no long change the values of the output register. That is, the output register is "write-once".

If we revise the rules for output register to the following new rule, then we can to circumvent the impossibility result and design BFT protocols for asynchronous networks. Each participant has an output register with values in $\{0, 1\}$. The output register starts with value 0. One can revise the value in the output register with a new value 0 or 1 only once. As soon the value in the output register is revised, it moves to the final decision state. In case that the value in the output register is never revised, we assume that the value 0 is the final decision value. From a first look, this new rule seems to be useless. But if we intepret the value 0 as an empty block and 1 as non-empty block in blockchains, then this rule is compatible with the blockchain eventual consistency principle. As we have already mentioned in the Introduction section, Andreina et al [1] noted taht "*blockchain protocols target eventual consistency, resp liveness, meaning that the chain of some honest party may be inconsistent, resp. stuck, during asynchronous periods but then it will catch up as soon as the network obeys a synchronous regime.*". With this new rule, we can design the following BFT protocol that tolerate $\lfloor \frac{n-1}{3} \rfloor$ Byzantine faults in fully asynchronous networks.

Our setting for the BFT protocol is similar to that in Wang [18]. Assume that there are n_0 clients $L_0, \cdots, L_{n_0-1}$ and $n = 3t + 1$ BFT participants $P_0, \cdots, P_{n-1}$. Each client L_i submits a proposal to the BFT participants $P_0, \cdots, P_{n-1}$. Since we do not assume a reliable multicast communication channel, L_i may submit different proposals to different BFT participants. Using a commonly agreed selection criteria, each BFT participant P_i selects a proposal p_i from the potentially n_0 submitted proposals. We call this proposal p_i as P_i's input proposal. The goal of the protocol is for BFT participants $P_0, \cdots, P_{n-1}$ to reach a consensus on one proposal p from all of the input proposals. Generally, we can use a robust threshold signature scheme such as Boneh-Lynn-Shacham (BLS) signatures [4] 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. The protocol proceeds as follows.

1. Each participant P_i sends his signed input proposal p_i to all participants. Since we do not assume a reliable multicast communication channel, P_i may send different proposals to different participants.

2. If a participant P_i collects $2t + 1$ digital signatures for some proposal p, it writes $\langle p, cert_i \rangle$ to her output register, multicasts $\langle p, cert_i \rangle$ to all participants and moves to the decision state where $cert_i$ is a collection of $2t + 1$ digital signatures on p.

3. In a synchronous network setting, if the participant P_i does neither receive $2t+1$ digital signatures for some proposal p nor receive a valid message $\langle p, cert_i \rangle$, then it writes EMPTY to her output register and moves to the decision state.
4. In a asynchronous network setting, before the participant P_i receives $2t+1$ digital signatures for some proposal p or receives a valid message $\langle p, cert_i \rangle$, it temporarily writes EMPTY to her output register and moves to the wait state (where the wait state could be considered as a final state also).

For the n_0 clients $L_1, \cdots, L_{n_0}$, if L_i is qualified to submit the final proposal, we call L_i the lead client. For safety analysis of the protocol, we can distinguish the following two cases.

1. The lead client is honest. In this case, all the BFT participants $P_1, \cdots, P_n$ received the same proposal p and the $2t+1$ honest participant will digitally sign p and multicast it to all participants. Thus all honest participants will agree on this proposal p at the end of the protocol run.
2. The lead client is malicious. In this case, if all BFT participants marked their output register as EMPTY, then the safety property is satisfied. Assume that at least one BFT participant P_i received $2t+1$ digital signatures on a proposal p. In this case, at least $t+1$ honest participant signed this proposal. Thus no honest participant would receive another proposal p' with $2t+1$ valid digital signatures. Since P_i would then multicast $\langle p, cert_i \rangle$ to all participants, all honest participants will write $\langle p, cert_i \rangle$ to the output register.

The liveness property of the protocol could be proved assuming that honest clients will be qualified to submit a proposal at least 50% of the time.

References

[1] S. Andreina, J.-M. Bohli, G.O. Karame, W. Li, and G.A. Marson. Pots-a secure proof of tee-stake for permissionless blockchains. *IACR Cryptology ePrint Archive* `https://eprint.iacr.org/2018/1135.pdf`, 2018:1135, 2018.

[2] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino. State machine replication in the Libra Blockchain. `https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf`, Aug 7, 2018.

[3] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proc. 2nd ACM Annual Symposium on Principles of Distributed Computing, Montreal, Quebec*, pages 27–30, 1983.

[4] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *Journal of cryptology*, 17(4):297–319, 2004.

[5] Vitalik Buterin. A guide to 99% fault tolerant consensus. `https://vitalik.ca/general/2018/08/07/99_fault_tolerant.html`, Aug 7, 2018.

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

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

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

[9] 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.

[10] Richard M Friedberg. Two recursively enumerable sets of incomparable degrees of unsolvability (solution of post's problem, 1944). *Proc. the National Academy of Sciences of the USA*, 43(2):236, 1957.

[11] 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.

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

[13] Albert A Muchnik. On the unsolvability of the problem of reducibility in the theory of algorithms. In *Dokl. Akad. Nauk SSSR*, volume 108, page 1, 1956.

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

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

[16] 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.

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

[18] Yongge Wang. Another look at algorand. arXiv:1905.04463v1, 2019. `https://arxiv.org/abs/1905.04463`.

[19] 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.

6 Appendix: DLS BFT protocol

In this section, we review the DLS BFT protocol by Dwork, Lynch, and Stockmeyer [8]. DLS first proposed the basic round BFT model and the revised it to various semi-asynchronous models. In the basic round BFT protocol, we assume that all messages after the unknown step GST (Global Stabilization Time) will be delivered in the same step that the message was sent and loss of a message before the GST does not necessarily make the sender or the receiver gaulty. Furthermore, though all participants have a common numbering for the round, they do not know when the step GST occurs.

During the protocol run, each participant P_i maintains a local variable PROPER$_i$ which contains a set of values that the participant knows to be proper. At the start of the protocol, PROPER$_i$ contains the inital value of the participant P_i. Each participant attaches its PROPER$_i$ to every message it sends. If a participant P_i has ever received $2t+1$ values from $2t+1$ participants and there are no $t+1$ identical values, it will add all potential values to its PROPER$_i$. If a participant P_i receives $t+1$ identical value v from $t+1$ participants, then it it adds v to its PROPER$_i$. The protocol proceeds from step to step. For step s, the participant P_i is called the leader if $s \equiv i \mod n$. Each step $s \geq 0$ contains the following four sub-steps (where we assume that P_i is the leader for step s):

1. Each participant P_j (including P_i) sends the digitally signed message $sig_j(L_j, s)$ to the leader participant P_i where L_j is a list of all its acceptable values that are in the set PROPER$_j$.

2. If the leader P_i receives an acceptable value v in sub-step 1 from $2t+1$ participants, P_i **broadcasts** the signed message $sig_i(\text{lock } v, s, \text{proof})$ to all participants where proof is a list of signed messages showing that v is acceptable to at least $2t+1$ participants.

3. For each participant P_j, if it receives a valid message $sig_i(\text{lock } v, s, \text{proof})$ from sub-step 2, then it locks the value v by recording the valid lock $sig_i(\text{lock } v, s, \text{proof})$. P_j sends an acknowledgement to the leader P_i. If P_i receives at least $2t+1$ acknowledgements, then P_i *decides* on the value v and continues to participate in the protocol.

4. Every participant P_j **broadcasts** its locked value $sig_{i'}(\text{lock } v', s', \text{proof})$ to all participants if it has any locked value. A participant will release its lock on a value $sig_{i''}(\text{lock } v'', s'', \text{proof})$ if it receives a lock $sig_{i'}(\text{lock } v', s', \text{proof})$ with $s' \geq s''$ and $v' \neq v''$.

Dwork, Lynch, and Stockmeyer [8] extends the above basic round BFT protocol to partially synchronous communication networks (with synchronous participants) as follows.

- Upper bounds Δ holds eventually. Let $R = n + \Delta$. Each sub-step in the basic round BFT protocol is simulated by R sub-sub-steps. Specifically, the participant uses the first n sub-sub-steps to send its sub-step messages to the n participants (sending to one participant at a time) and uses the last Δ sub-sub-steps to perform message receive operations. The state transition for the sub-step is simulated at the last sub-sub-step.

- Δ is unknown. Let $R_s = n + s$ for $s > 0$. Each sub-step in the basic round BFT protocol step s is simulated by R_s sub-sub-steps as in the case where Δ holds eventually.