Byzantine Fault-Tolerant Consensus

Fatemeh Zarinjouei

December 5, 2024

Abstract

Consensus algorithms play a pivotal role in distributed systems. In such environments, where nodes collaborate to achieve agreement, the potential presence of Byzantine nodes poses a significant challenge, as they can disrupt the consensus process. Consequently, the ability to tolerate Byzantine faults while successfully reaching consensus is a fundamental requirement. To date, numerous Byzantine Fault Tolerant (BFT) algorithms have been developed, alongside various methods to enhance their performance. This report provides a survey of three prominent algorithms: PBFT, HotStuff, and HotStuff-2. It compares these protocols in terms of their algorithmic steps, communication complexity, and the number of phases required for consensus.

1 Introduction

In distributed systems, achieving agreement among various nodes is a fundamental challenge essential for ensuring the system operates coherently and without error. Consensus algorithms serve as key components of these systems, playing a decisive role in maintaining data consistency, reliability, and security. In such environments, the system state can fluctuate due to network delays, potential failures, and malicious behavior by certain nodes. Without proper management, these factors can lead to data inconsistency or loss. The objective of consensus algorithms is to ensure that correct (healthy) nodes agree on a specific order of command execution, enabling the system to continue its operations correctly and with high reliability [22]. With the growing interest in blockchain technology, consensus algorithms have garnered increased attention, as they constitute a foundational component of blockchains. In this context, a critical concept is Byzantine Fault Tolerance (BFT), which refers to a system's ability to maintain correct operation even in the presence of malicious behavior or arbitrary faults within some nodes. In the State Machine Replication (SMR) model [12, 19], system nodes coordinately maintain a shared state. A BFT SMR protocol guarantees that even if f out of n total nodes are Byzantine, the correct nodes can agree on the command execution order and produce identical results.

The communication model employed in many consensus protocols, particularly those examined in this report, is the Partially Synchronous Model. In this model, after an unknown synchronization period known as the Global Stabilization Time (GST), message delays become bounded by a specific constant Δ [8]. Under these conditions, it has been proven that to achieve consensus, the total number of nodes must satisfy $n \geq 3f + 1$ [9]. The mechanism that enables a new leader to collect information and broadcast it to nodes is termed the View-Change algorithm; this constitutes the core of the replication process. However, prior to recent literature, it was widely believed that a View-Change based on a two-phase paradigm was not straightforward [17], was highly error-prone [1], and incurred significant communication costs even in moderately sized systems. This process entails the new leader collecting information from $n - f$ nodes, such that each node reports its highest observed Quorum Certificate (QC).

The first practical protocol implementing BFT SMR in the partially synchronous model was the PBFT algorithm [5]. It featured a two-phase structure where the leader could achieve agreement within two message-transmission rounds per consensus instance. However, this protocol exhibited a worst-case communication complexity of $O(n^3)$, which was later reduced to $O(n^2)$ through the use of threshold signatures [20].

In this report, following an exposition of preliminary concepts and the problem definition, the following three BFT-based consensus algorithms are introduced sequentially:

- **PBFT**: The first practical protocol for BFT SMR, featuring a message complexity of $O(n^2)$ and executing in two phases [5].
- **HotStuff**: An improved variant of PBFT that reduces message complexity to $O(n)$ but executes in three phases [20].
- **HotStuff-2**: A variant of HotStuff that maintains $O(n)$ message complexity while executing in only two phases [15].

After a detailed description of all three algorithms, we provide a comparative analysis to elucidate the advantages and disadvantages of each.

2 Preliminaries

In this section, we review the fundamental concepts and prerequisites necessary for understanding Byzantine consensus algorithms.

Global Stabilization Time (GST): Global Stabilization Time (GST) is a point in time after which the network guarantees that message delays remain within a known bound Δ. Prior to GST, message delays may be unbounded; however, after this time, it is assumed that network communication stabilizes, enabling consensus algorithms to make progress [8].

Partially Synchronous Model: In this model, message delays are bounded by a specific constant Δ only after the Global Stabilization Time (GST) has passed [8]. The consensus algorithms examined in this report are designed within the partially synchronous model.

Signature and Authentication Mechanisms in BFT : To prevent message forgery and reduce communication complexity, BFT protocols employ digital signatures and threshold signatures. PBFT utilizes traditional signatures, which results in a communication complexity of $O(n^3)$ (specifically during view-change). In contrast, HotStuff employs threshold signatures to reduce this complexity to $O(n^2)$ [10].

Optimistic Responsiveness Formally, a Byzantine consensus protocol is optimistically responsive if, during periods of low network delay and honest behavior, it can reach a final decision without waiting for a predetermined maximum time bound, depending solely on receiving messages from $2f + 1$ correct nodes [20].

The following concepts are frequently used throughout these consensus algorithms:

- *Quorum Certificate (QC)*: A proof indicating that at least $2f + 1$ nodes have voted for a specific value.
- *View*: A monotonically increasing number that identifies the current leader; this number is incremented in the event of leader failure.
- *View Change*: A mechanism through which the system elects a new leader if the current leader exhibits faulty or poor performance.

3 Problem Definition

In a distributed system, nodes are interconnected via a communication network. This network is prone to message delays, duplication, and out-of-order delivery [8]. Furthermore, certain nodes may be malicious and exhibit Byzantine behavior due to various factors such as cyber-attacks or software bugs. These malicious nodes are capable of arbitrary deviations from the protocol, including transmitting conflicting messages, omitting messages, or colluding with other faulty nodes [5]. The Byzantine failure model posits that the system can tolerate up to f malicious nodes, provided that the total number of nodes satisfies $n \geq 3f + 1$ [2]. This condition ensures that a quorum of non-faulty nodes $(2f + 1)$ remains, enabling the distinction between correct and malicious behaviors. The primary objective of the Byzantine Fault Tolerance problem is to guarantee two key properties: Safety and Liveness.

Safety : The Safety property ensures that the system behaves like a centralized implementation, even in the presence of malicious nodes [13]. This property comprises two main aspects:

- **Agreement**: All non-faulty nodes must agree on the order and result of operations [19].
- **Consistency**: All non-faulty clients must observe a consistent system state [5].

Formally, if two non-faulty nodes view values v and v' as the result of a request execution, it must hold that $v = v'$.

Liveness : The Liveness property guarantees that if a correct client issues a request, it will eventually receive a valid response [8]. This property requires the following conditions to be met:

- The network eventually reaches a stable state where message delays are bounded (the **Partially Synchronous** model) [8].
- A quorum of nodes $(2f + 1)$ remains non-faulty and continues the consensus process [5].

Formally, if a correct client sends a request, it must receive a valid response within a finite time after the Global Stabilization Time (**GST**) is reached [2]. Furthermore, to achieve the aforementioned goals, Byzantine Fault Tolerant algorithms also possess the property of Optimality [4].

Optimality : This property ensures that safety and liveness properties can be maintained against Byzantine faults with a minimum of $n = 3f + 1$ nodes [2]. This is the minimal number of nodes required to allow the system to operate correctly in the presence of up to f malicious nodes.

In the following sections, we examine three significant algorithms that address this problem: **PBFT**, **HotStuff**, and **HotStuff-2**. Each of these algorithms ensures that non-faulty nodes can agree on a common value, even if up to $\frac{n-1}{3}$ of the total n nodes in the system exhibit malicious or arbitrary behavior.

4 Practical Byzantine Fault Tolerance (PBFT)

This algorithm [5] employs state machine replication and ensures agreement through three primary phases: *pre-prepare*, *prepare*, and *commit*. The details of the algorithm are presented below.

4.1 The Basic Algorithm

Initially, a client sends a request containing the desired operation and a timestamp to the primary node (leader). The timestamp ensures that the request is executed exactly once. The primary assigns a sequence number to the received request and multicasts it, along with the current view number and the sequence number, to all backup nodes (replicas). This phase, known as the *pre-prepare* phase, serves as the starting point for the consensus process. The message broadcast by the primary includes the current view number, the sequence number, and the digest (hash) of the client's request.

In the *prepare* phase, upon receiving the message from the primary, backup nodes verify the digital signature and the validity of the sequence number. If the message is valid, the nodes multicast a *prepare* message to all other nodes. In this phase, each node collects prepare messages to ensure that the request is acknowledged by a sufficient number of nodes ($2f$). This step allows non-faulty nodes to agree on a total order of requests within the current view. Upon completing the prepare phase, nodes enter the *commit* phase. In this phase, nodes multicast a *commit* message to all other nodes. Once a node receives at least $2f + 1$ commit messages that match the initial request, it executes the requested operation and sends the result to the client. The client waits for $f + 1$ matching replies from different nodes; if it does not receive sufficient replies within a specified timeout, it retransmits the request.

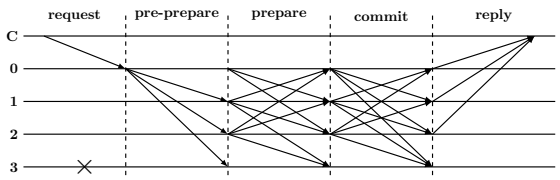

Figure 1: Normal Execution of PBFT [5]

4.2 Garbage Collection

This section examines the mechanism for discarding executed messages from node logs to prevent unbounded storage growth. To maintain safety, messages must be retained in a node's log until the node can prove to others during a view change that the corresponding requests have been executed by at least $f + 1$ non-faulty nodes. Additionally, if a node misses messages that have been discarded by all non-faulty nodes, it requires a method to retrieve the state.

Therefore, nodes need a mechanism to prove the correctness of their state. Generating such a proof after every operation is computationally expensive; thus, proofs are generated periodically. The states produced are termed *checkpoints*, and a checkpoint possessing a proof of correctness is called a *stable checkpoint*.

The proof of correctness for a checkpoint is generated as follows: When a node produces a checkpoint, it multicasts a *checkpoint* message to other nodes. This message contains the sequence number of the last request reflected in the state and the digest of the state. Each node collects these checkpoint messages until it receives $2f + 1$ messages with the same sequence number and digest. These $2f + 1$ messages constitute the proof of correctness. Once a checkpoint becomes stable, the node discards all *pre-prepare*, *prepare*, and *commit* messages with sequence numbers less than or equal to the checkpoint from its log. Older checkpoints and their associated messages are also discarded.

4.3 View Change Mechanism

If the current primary fails or behaves maliciously, the *view change* mechanism is triggered. This mechanism allows a new leader to be elected, thereby preserving liveness. Specifically, when a backup node detects that the primary has failed to progress requests, it starts a timer. If the timer expires and the request remains unexecuted, the backup initiates the view change process and multicasts a *view-change* message to all other nodes. This message includes:

1. The sequence number of the last stable checkpoint stored by the node.

2. The proof for this stable checkpoint.

3. A set of requests that have prepared in the current view but have not yet executed.

For every request in this set, a valid *pre-prepare* message and two valid *prepare* messages signed by different nodes are included.

The new primary waits to receive at least $2f+1$ valid *view-change* messages from other nodes before initiating the new view. The new primary broadcasts a *new-view* message to all nodes. This message includes the new view number $(v+1)$, the set of *view-change* messages received, and a set of *pre-prepare* messages for requests that must be processed in the new view. Requests that prepared in the previous view but did not execute are re-processed in the new view. To calculate the set of *pre-prepare* messages, the new primary performs the following steps:

1. Determines the sequence number min-s of the latest stable checkpoint in the received messages.

2. Determines the highest sequence number max-s in the received *prepare* messages.

3. Creates a new *pre-prepare* message for every sequence number between min-s and max-s.

The new primary then stores these *pre-prepare* messages in its log. Following these steps, the node enters the new view $v+1$ and becomes ready to accept messages for it.

Upon receiving the *new-view* message, each backup node verifies its validity. This involves checking the signature, ensuring the set of *view-change* messages matches the local information, and verifying the validity of the request order specified in the *new-view* message. If the message is valid, the node enters the new view and processes the *pre-prepare* messages for the requests specified. After entering the new view, the algorithm proceeds as normal, and messages between min-s and max-s are processed.

In some instances, a node may be missing request messages or the stable checkpoint, as these are not transmitted directly in the *new-view* message. In such cases, the node can request the missing information from other nodes. For example, a node can retrieve the state of a stable checkpoint from one of the nodes that previously broadcast valid checkpoint messages for it. Since at least $f+1$ of these nodes are non-faulty, the requesting node can always retrieve a valid stable checkpoint.

4.4 Communication Complexity Analysis

The communication complexity of PBFT under normal operation is analyzed as follows: In the *pre-prepare* phase, the primary sends a message to $n-1$ other nodes, resulting in $O(n)$ messages. In the *prepare* phase, every node multicasts its message to all other nodes, leading to $O(n^2)$ messages. Similarly, in the *commit* phase, every node again multicasts a message to all others, which also incurs $O(n^2)$ complexity. Therefore, the total communication complexity in the normal case is $O(n^2)$.

During the view change process, communication complexity can increase from $O(n^2)$ to $O(n^3)$. The primary driver for this increase is the retransmission of messages related to requests that prepared in the previous view but did not execute. In this scenario, the new primary must create a new *pre-prepare* message for each of these requests and broadcast them to all nodes. In the worst case, the number of such requests can be proportional to the number of nodes $(O(n))$, and since each message is sent to all nodes, the complexity of this phase becomes $O(n^3)$. This high complexity is considered one of the scalability weaknesses of PBFT in large systems.

5 HotStuff

This algorithm [20] consists of four primary phases: *prepare*, *pre-commit*, *commit*, and *decide*. Each phase serves a specific objective and plays a crucial role in ensuring protocol safety and liveness. A pivotal element of this protocol is the aggregation of $n-f$ votes from nodes for a leader's proposal, termed a Quorum Certificate (QC). A QC is associated with a specific node and a view number, employing a threshold signature scheme to combine these partial signatures into a single vote.

The algorithm executes through the following four phases:

- **Prepare Phase:** Every view begins with the new leader receiving at least $n-f$ *new-view* messages. These messages contain the highest *PrepareQC* observed by each node. The leader processes these messages and selects the branch extending from the highest view where a *PrepareQC* was formed. This certificate is known as the *highQC*. Since *highQC* is the highest among $n-f$ nodes, no higher view could have reached a commit decision. Thus, the branch extending from the *highQC* node is considered safe. The leader then creates a new proposal containing a branch of command history, extending the node specified by *highQC*. This new node, which includes a hash of its parent, is broadcast to all nodes in a *prepare* message along with the *highQC*. Upon receiving the *prepare* message, nodes verify the safety of the new proposal using the *safeNode* predicate. If the proposal is accepted, the node sends a partial signature vote to the leader.

- **The *safeNode* Predicate:** The *safeNode* predicate is a fundamental component of this protocol. It examines a proposed message m carrying a justification QC (*m.justify*) to determine if the node *m.node* is safe to accept. The safety rule states that the branch extending from *m.node* must be an extension of the currently locked node, *lockedQC.node*. In other words, a new node is acceptable only if it is built upon a branch extending from the node previously locked by the replica. Alternatively, the liveness rule dictates that a node accepts proposal m if the view number of *m.justify* is higher than the view number of the current *lockedQC*. This rule allows nodes to transition to newer views, thereby preventing potential deadlocks. The predicate evaluates to *True* if at least one of these two rules is satisfied.

- **Pre-Commit Phase:** Upon receiving at least $n-f$ votes from the nodes, the leader combines these partial signatures using threshold signatures to generate a new Quorum Certificate (*prepareQC*). The leader then broadcasts this certificate to all nodes in a *pre-commit* message. Upon receiving the *pre-commit* message, each node stores the QC in its local state and sends a *pre-commit* vote to the leader.

- **Commit Phase:** In this phase, the leader waits to receive at least $n - f$ *pre-commit* votes. After collecting these votes, the leader creates a new certificate called *PrecommitQC* and broadcasts it to all nodes. Upon receiving the *PrecommitQC*, a node responds by sending a *commit* message to the leader. Crucially, the node becomes **locked** on the *PrecommitQC* by setting its local *lockedQC* variable to equal the *precommitQC*.

- **Decide Phase:** Finally, upon receiving at least $n-f$ *commit* votes, the leader combines them using threshold signatures to form a final certificate, the *CommitQC*. This certificate is broadcast to all nodes in a *decide* message. Any node receiving the *decide* message marks the leader's proposal as the final consensus decision, executes the commands contained in the decided branch, and transitions to the next view.

Interrupt and View Change If a node fails to reach a conclusion in any phase due to message delays or leader misconduct, an interrupt occurs. In this scenario, the node increments its view number and sends a *new-view* message to the new leader.

5.1 Communication Complexity Analysis

In each phase, the leader broadcasts messages to all n nodes, and subsequently, each node sends a response to the leader. Consequently, the number of messages exchanged in each of these phases is of order $O(n)$. Additionally, in the *decide* phase, the leader sends a final message to the entire network, which also carries a complexity of $O(n)$. Therefore, under normal conditions where no faults occur, the total complexity per consensus round is $O(n)$. In scenarios where the leader is faulty or nodes fail to reach agreement, the *view change* process is executed. In this process, each node must transmit its current state to the new leader, and the new leader, after processing these messages, transmits new information to the nodes. In a single view change, the number of messages exchanged between nodes and the new leader is $O(n)$. In the worst-case scenario, if leader failures occur repeatedly within a short timeframe, the total number of messages exchanged scales to $O(n^2)$.

For the pseudocode of the protocol, please refer to Protocol **??** in the Appendix.

6 HotStuff-2

This algorithm [15] comprises two primary components: the steady-state protocol and the **Pacemaker** protocol for advancing views. The steady-state protocol executes final decisions within a single view when all correct nodes overlap in that view for a sufficient duration and the designated leader for that view is correct.

6.1 Data Structures and Terminology

This section outlines the data structures and terminology specific to this algorithm.

Block Format: The protocol forms a chain of values. Each value in the chain is termed a block. Every block B_k at height k consists of $B_k := (b_k, h_{k-1})$ where b_k is the proposed value at height k, and $h_{k-1} := H(B_{k-1})$ is the hash of the preceding block. The genesis block is $B_1 = (b_1, \perp)$ with no predecessor. Each subsequent block B_k must identify its predecessor using the hash.

Block Extension and Conflict: A block B_l is said to extend block B_k if B_k is an ancestor of B_l (where $l > k$). Two blocks B_l and B'_l are in conflict if they are not identical and neither extends the other.

Certificates: In the protocol, nodes vote for blocks by signing them. A set of signatures from $2t + 1$ nodes in view v for block B_k is known as a certificate, denoted as $C_v(B_k)$.

Timeout Certificates: If progress is not made within a view, nodes may stop the timer and utilize the

Pacemaker to transition to the next view. A TC_v is a certificate generated by $2t + 1$ nodes to enter view v.

Locked Blocks: At any given moment, a node locks the highest certified block it is aware of. These locks are utilized to maintain the safety of final decisions throughout the protocol's execution.

6.2 Steady-State Protocol

In each view, a specific leader is responsible for driving the consensus of a block sequence. Leaders can be selected statically or via more complex randomized techniques. For simplicity, it is assumed that leaders are selected via *Round-Robin*, such that the leader of view v is node ($v \mod n$). The goal of a leader in a view is to extend the highest certified block in the current sequence with a new block and get it certified. This process is executed as follows:

1. **Propose:** The leader in view v creates a block B_k at height k, containing the proposed value, the view number, and the highest certificate known to the leader. The leader then broadcasts the proposed block to all nodes.

2. **Vote:** Each node maintains the highest certificate it has received thus far. The node votes for block B_k and sends its vote to the leader only if this block extends the highest certificate known to the node.

3. **Certify:** A block becomes certified when $2t + 1$ nodes vote for it. The certificate $C_v(B_k)$ is aggregated by the leader.

4. **Second Vote Round:** The leader conducts another round of voting, but nodes send their votes to the leader of view $v + 1$.

The pseudocode for the "View Protocol" is provided in Algorithm 1.

6.3 Pacemaker

The Pacemaker is responsible for synchronizing nodes to enter views. Progress to view $v + 1$ is achieved via one of two methods:

- **Case 1:** If $C_v(C_v(B_k))$ (a double certificate) is obtained, the Pacemaker can immediately transition to the next view. This guarantees that optimal performance is maintained under optimistic conditions—i.e., when there are no faults and the network is synchronous. In this case, since the certificate is broadcast by the leader, the Pacemaker requires no additional action to synchronize entry into the view.

- **Case 2:** If such a certificate is not obtained, the Pacemaker must wait for at least $\tau = \Omega(\Delta)$ to allow a correct leader in synchronous conditions to make progress. This delay is essential for maintaining protocol liveness, as it provides correct leaders sufficient time to drive progress [2]. When τ timers expire at a sufficient number of nodes, the Pacemaker can proceed to the next view and coordinate nodes to enter the new view approximately simultaneously.

Various methods exist for view synchronization; however, the authors sought a method possessing the property of *optimistic responsiveness*. The description below adapts the view synchronization method found in [7, 14]. In this method, consecutive views are grouped into *epochs*, such that each epoch contains $t + 1$ consecutive views. Nodes utilize a synchronization protocol similar to *Cogsworth* [18] in the first view of each epoch, and subsequently use timers to progress through the remaining views of the epoch if the base consensus protocol fails to make progress. In the worst case, the communication complexity is $O(n^2)$ per decision, with a delay equivalent to $O(n\Delta)$. The Pacemaker protocol is described in Algorithm 2.

6.4 View Change Mechanism

This mechanism ensures that a correct leader can form a certified block in a view after GST. It is integrated into the Pacemaker such that a certified block is provided to the new leader with linear communication cost. The process operates as follows:

- **Case 1:** The new leader has already received the certified block.

- **Case 2:** Progressing to a view without forming a previous certificate incurs a mandatory delay of $\tau = \Omega(\Delta)$. The leader can detect that it is in Case 2. Since all nodes enter the view within $O(\Delta)$ time, the leader can wait for $\tau + \Delta$ to ensure all correct nodes respond before making a proposal.

In the pessimistic case, this view change mechanism imposes an additional delay of 2Δ on top of $\tau = O(\Delta)$ compared to the basic two-phase HotStuff protocol.

6.5 Communication Complexity Analysis

The communication complexity of the HotStuff-2 algorithm can be analyzed in two scenarios:

1. **Optimistic Case:** Under conditions where the leader is correct and the network operates synchronously, each decision is reached with a communication complexity of $O(n)$. This is achieved

Algorithm 1 View Protocol (Per replica in view v)

Setup: Let v be the current view number and L_v be the leader in view v.

Phase 1: Enter
if entering view v using $C_{v-1}(C_{v-1}(B_{k-1}))$ **then**
    Proceed directly to **Propose** step.
else
    Set a timer $P_{pc} + \Delta$, then proceed to **Propose** step.
    Send the locked certificate to L_v. ▷ Replica Action
end if

Phase 2: Propose
if Node is Leader L_v **then**
    Let $B_k := (b_k, h_{k-1})$ extending highest certified block B_{k-1}.
    Broadcast $\langle \text{propose}, B_k, v, C_{v'}(B_{k-1}), C_{v''}(C_{v''}(B_{k''}))\rangle_{L_v}$.
end if

Phase 3: Vote and Commit
On receive valid proposal $\langle \text{propose}, B_k, v, C_{v'}(B_{k-1})\rangle_{L_v}$:
if $C_{v'}(B_{k-1})$ is ranked $\geq$ locked block **then**
    Send $\langle \text{vote}, B_k, v\rangle$ (threshold signature share) to L_v.
    Update lock to B_{k-1} and certificate to $C_{v'}(B_{k-1})$.
    Commit block $B_{k''}$ and all its ancestors.
end if

Phase 4: Prepare
if Node is Leader L_v **and** received $2t + 1$ votes for B_k **then**
    Form certificate $C_v(B_k)$.
    Broadcast $\langle \text{prepare}, C_v(B_k)\rangle_{L_v}$.
end if

Phase 5: Vote2
On receive $\langle \text{prepare}, C_v(B_k)\rangle_{L_v}$:
Update lock to B_k and locked certificate to $C_v(B_k)$.
Send $\langle \text{vote2}, C_v(B_k), v\rangle$ to Leader L_{v+1}.

through the use of threshold signatures, which compress messages and reduce the volume of inter-node communication.

2. **Worst Case:** In the presence of faulty leaders or synchronization issues, the protocol may proceed through up to $t + 1$ views before reaching a correct leader. This results in an aggregate communication complexity of $O(n^2)$.

7 Comparison

7.1 Overview of Approaches

All three protocols—PBFT, HotStuff, and HotStuff-2—employ a nearly identical approach to guarantee the safety property. In each view, these protocols define a voting phase to ensure the uniqueness of a value among nodes, followed by another phase to allow nodes to commit to a value. The two-phase regime that implements this approach is described below.

When a node commits to a value v after the second phase, it is guaranteed that at least $2t + 1$ nodes (and thus at least $t + 1$ correct nodes) have locked on value v in the first phase. These $t + 1$ correct nodes guard the safety of the commit; they will not vote for a conflicting value until proof is provided that voting for such a value is safe.

Consequently, all these protocols rely on a **Lock-Commit** paradigm: a sufficient number of nodes must be locked before any node can commit. Therefore, if the leader fails or the network stalls, the system may find itself in one of the following three states:

Algorithm 2 Pacemaker Protocol

Require: Node enters view v

Set Timers:
if $(v \mod t+1) = 0$ **then**
    Set view-k timer to expire at slots $k\tau$, for $k = 1 \dots (t+1)$.
end if
Proceed to **View Protocol** for view v.

On Timer Expiration:
Stop processing messages and voting for view v.
if $(v \mod (t+1)) \neq 0$ **then**
    Execute View Protocol for next view.
else
    Epoch synchronization: Perform "epoch-view" procedure for view v.
    Send timeout message $\langle wish, v+1 \rangle$ to the $t+1$ leaders in epoch.
    if Leader collects $2t+1$ $\langle wish, v+1 \rangle$ messages **then** ▷ Forms TC_{v+1}
        Broadcast TC_{v+1} to all parties.
    end if
end if

Advance View:
if Received $TC_{v'}$ for $v' > v$ **and** $(v' \mod (t+1) = 0)$ **then**
    Relay TC to $t+1$ leaders in the epoch of v'.
end if
if Received $C_{v'-1}(C_{v'-1}(B_{k'}))$ **then**
    Advance to the next view.
end if

1. No node is locked; thus, no node has committed to any value.
2. One or more nodes are locked on a value, but no correct node has committed.
3. $2t+1$ nodes are locked, and some (but not necessarily all) correct nodes have committed.

In the first case, all nodes are free to vote for other values in subsequent views, so no issue arises. The key question concerns how nodes locked on a value should behave in the other two cases, given that they do not know the global system state. It has been shown that in all these safety-oriented protocols, locked nodes guarantee the safety of a commit. By default, they assume the system is in the third state (commit occurred) and will not vote for a conflicting value unless evidence to the contrary is presented.

However, when the system is in the second state, a mechanism is required to ensure that locked correct nodes can vote for a safe proposal from the new leader after a view change, even if it conflicts with their lock. The discussed schemes differ in how they address this. Since these are safety-favored protocols, we are primarily concerned with the progress of a correct leader, as a malicious leader can, at worst, delay progress (e.g., by not sending a proposal).

7.2 Differences in View Change Mechanisms

7.2.1 PBFT

In the PBFT protocol, the leader collects lock statuses from $2t+1$ nodes at the beginning of every view. For example, in the third state described above, some nodes have committed in previous views. Among the $2t+1$ locks in the status, up to t may be from faulty nodes, and up to t may be from correct nodes that are not locked on the committed value. However, there is at least one correct node locked on the committed value. This correct node acts as a guardian of the commit's safety and will not vote for a leader's proposal that conflicts with its lock. Furthermore, the leader is guaranteed to observe this lock when collecting the status.

Consider the second state, where no node committed in the previous view, but some are locked. Their status might not be part of the quorum collected by the leader (e.g., the leader might receive status from all nodes ex-

cept the correct locked ones, and Byzantine nodes might conceal their locks). In this specific case, the leader is guaranteed that no correct node has committed and is effectively free to propose any value. To convince the correct nodes that are locked on a conflicting value, the leader includes a "status certificate" containing $2t + 1$ lock statuses in its proposal. Essentially, the leader provides proof of the absence of a lock from itself and other nodes. This status certificate provides sufficient information for all correct nodes to vote for the proposal.

Sending this message can be **responsive**, as the leader can act as soon as it receives $2t+1$ status messages. However, the status certificate has linear size, and including it in the proposal results in quadratic communication complexity.

7.2.2 HotStuff

HotStuff addresses the liveness problem differently while ensuring that the value corresponding to the highest lock is proposed by the leader. For safety, the rule is: "If a node commits to a value, then at least $2t + 1$ nodes are locked on that value. This lock guarantees the safety of the commit and ensures the next leader receives it."

To maintain liveness, HotStuff introduces an additional voting phase. With this new phase, the following statement always holds: "If a node is locked on a value, then at least $2t+1$ nodes are aware of this lock and possess a corresponding key (the Quorum Certificate). This key is shared with the next leader to effectively decide on a proposal."

Consider a scenario where a leader failure or weak network leaves some nodes locked at a critical moment; this extra phase guarantees that $2t + 1$ nodes have received the key corresponding to this lock. Consequently, the next leader, via the $2t+1$ keys it receives, becomes aware of the highest lock and will coordinate with the proposal having the highest global lock held by any correct node. Thus, HotStuff achieves linear view change and responsiveness, but the introduction of the key-distribution phase transforms it into a three-phase protocol.

7.2.3 HotStuff-2

HotStuff-2 approaches the liveness problem from a new perspective, posing the question: Is an additional phase truly necessary to solve this problem while retaining linear communication complexity and optimistic responsiveness? As discussed, if the leader is aware of the highest lock in the system and can convince all correct nodes of it, the problem is solved. The insight of this paper is that the new leader can choose between two options:

1. If the leader receives a lock (QC) from the *immediate* previous view, it is certain that it has obtained the highest locked value in the system. In this case, the leader can proceed with a proposal in a **responsive** manner.

2. Otherwise, the leader knows that a mandatory time delay of $\Omega(\Delta)$ must have elapsed in the previous view. In this case, responsiveness is already lost. Therefore, the leader waits an additional $O(\Delta)$ time to ensure all nodes have entered the new view before proceeding. This wait ensures it obtains the highest locked value in the system.

Essentially, if the leader receives a lock from the previous view, no higher lock can exist. Thus, a proposal respecting this lock will be accepted by all correct nodes. If the leader does not receive such a lock, by waiting for $O(\Delta)$, it receives lock information from all correct nodes. This situation arises if the previous leader was malicious or if the previous view occurred before GST. However, responsiveness cannot be expected in these cases anyway.

Therefore, if there is a chain of correct leaders after GST, each guarantees that the system progresses responsively and generates a certificate in the view that assists the next leader. In this way, all leaders (except potentially the first in the chain) can drive correct nodes to commit responsively.

For example, in HotStuff-2, one of two scenarios may occur: 1. Some nodes may not commit in a view but become locked; the highest lock is obtained by the leader of the next view, and the view proceeds responsively. 2. No (correct) node locks on a value in a view, and the leader of the next view is forced to wait Δ time to make a proposal.

It is worth noting that utilizing a dual regime to distinguish between normal/faulty view states has been employed in prior works. The first instance was Pala [6], which introduced a dual regime for the Tendermint protocol [3]. Subsequently, several works attempted to improve the number of phases in HotStuff [11]. The modifications in HotStuff-2 are remarkably simple and add no significant complexity to the original HotStuff protocol [16]. The authors aimed to demonstrate that a simple solution is sufficient to simultaneously achieve linear view change, optimistic responsiveness, quadratic worst-case complexity, and two-phase commit.

Table 1 compares these three algorithms regarding four properties: number of phases, worst-case communication complexity, view change communication complexity, and optimistic responsiveness.

8 Conclusion

In this report, we examined three prominent consensus algorithms: *PBFT*, *HotStuff*, and *HotStuff-2*. Overall,

Table 1: Comparison of PBFT, HotStuff, and HotStuff-2

Algorithm	Phases	Worst-Case Comm.	View-Change Comm.	Responsive VC
PBFT	2	$O(n^3)$	$O(n^2)$	Yes
HotStuff	3	$O(n^2)$	$O(n)$	Yes
HotStuff-2	2	$O(n^2)$	$O(n)$	Yes

PBFT is recognized as a seminal algorithm with a foundational structure that continues to serve as a baseline for numerous subsequent studies. *HotStuff* introduced a novel approach by significantly reducing communication complexity and enhancing efficiency. Building upon this, *HotStuff-2* continued the evolutionary trajectory of this family of algorithms by reducing the number of execution phases required by *HotStuff*.

Experimental evaluations indicate that *HotStuff-2* exhibits superior performance in low-latency environments with a minimal number of Byzantine nodes; however, in scenarios involving a higher density of Byzantine faults, *HotStuff* maintains a relative performance advantage [21]. In summary, while *PBFT* offers robust safety guarantees, it is limited by poor scalability. In contrast, *HotStuff* and *HotStuff-2*, characterized by their linear communication complexity, represent significantly more efficient and scalable solutions for Byzantine consensus.

References

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.

[2] Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires t+1 rounds. *Information Processing Letters*, 71(3):155–158, 1999.

[3] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[4] Christian Cachin. Blockchain - from the anarchy of cryptocurrencies to the enterprise (keynote abstract). In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, volume 70 of *LIPIcs*, pages 2:1–2:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[6] T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptol. ePrint Arch.*, page 981, 2018.

[7] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine consensus is θ (n 2): the dolev-reischuk bound is tight even in partial synchrony! *Distributed Computing*, 37(2):89–119, 2024.

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

[9] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

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

[11] Mohammad M. Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and robust BFT protocol for blockchains. *IEEE Trans. Dependable Secur. Comput.*, 21(4):2478–2493, 2024.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[13] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[14] Andrew Lewis-Pye. Quadratic worst-case message complexity for state machine replication in the partial synchrony model. *arXiv preprint arXiv:2201.01107*, 2022.

[15] Dahlia Malkhi and Kartik Nayak. Extended abstract: Hotstuff-2: Optimal two-phase responsive BFT. *IACR Cryptol. ePrint Arch.*, page 397, 2023.

[16] Dahlia Malkhi and Maofan Yin. Lessons from hotstuff. In *Proceedings of the 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*, pages 1–8, 2023.

[17] James Mickens. The saddest moment. *login Usenix Mag.*, 39(3), 2014.

[18] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 1(2), oct 22 2021. https://cryptoeconomicsystems.pubpub.org/pub/naor-cogsworth-synchronization.

[19] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[20] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Siyuan Zhao, Yanqi Wu, and Zheng Wang. Hotstuff-2 vs. hotstuff: The difference and advantage. *arXiv preprint arXiv:2403.18300*, 2024.

[22] Weiyu Zhong, Ce Yang, Wei Liang, Jiahong Cai, Lin Chen, Jing Liao, and Naixue Xiong. Byzantine fault-tolerant consensus algorithms: A survey. *Electronics*, 12(18), 2023.

A Detailed Protocols

In this appendix, we provide the detailed pseudocode for the Basic HotStuff protocol referenced in Section 5.

Algorithm 3 Basic HotStuff Protocol

Loop:
for $curView \leftarrow 1, 2, 3, \ldots$ **do**

▷ **Phase 1: Prepare**
As Leader:
Wait for $(n - f)$ NEW-VIEW messages.
$M \leftarrow \{m \mid \text{MATCHINGMSG}(m, \text{NEW-VIEW}, curView - 1)\}$
$highQC \leftarrow \arg\max_{m \in M}\{m.justify.viewNumber\}.justify$
$curProposal \leftarrow \text{CREATELEAF}(highQC.node, \text{client's command})$
Broadcast $\text{MSG}(\text{PREPARE}, curProposal, highQC)$
As Replica:
Wait for message m from leader$(curView)$ such that $\text{MATCHINGMSG}(m, \text{PREPARE}, curView)$.
if $m.node$ extends from $m.justify.node \wedge \text{SAFENODE}(m.node, m.justify)$ **then**
    Send $\text{VOTEMSG}(\text{PREPARE}, m.node, \perp)$ to leader$(curView)$
end if

▷ **Phase 2: Pre-commit**
As Leader:
Wait for $(n - f)$ votes $V \leftarrow \{v \mid \text{MATCHINGMSG}(v, \text{PREPARE}, curView)\}$.
$prepareQC \leftarrow \text{QC}(V)$
Broadcast $\text{MSG}(\text{PRE-COMMIT}, \perp, prepareQC)$
As Replica:
Wait for message m from leader$(curView)$ matching QC PREPARE.
$prepareQC \leftarrow m.justify$
Send $\text{VOTEMSG}(\text{PRE-COMMIT}, m.justify.node, \perp)$ to leader$(curView)$.

▷ **Phase 3: Commit**
As Leader:
Wait for $(n - f)$ votes $V \leftarrow \{v \mid \text{MATCHINGMSG}(v, \text{PRE-COMMIT}, curView)\}$.
$precommitQC \leftarrow \text{QC}(V)$
Broadcast $\text{MSG}(\text{COMMIT}, \perp, precommitQC)$
As Replica:
Wait for message m from leader$(curView)$ matching QC PRE-COMMIT.
$lockedQC \leftarrow m.justify$
Send $\text{VOTEMSG}(\text{COMMIT}, m.justify.node, \perp)$ to leader$(curView)$.

▷ **Phase 4: Decide**
As Leader:
Wait for $(n - f)$ votes $V \leftarrow \{v \mid \text{MATCHINGMSG}(v, \text{COMMIT}, curView)\}$.
$commitQC \leftarrow \text{QC}(V)$
Broadcast $\text{MSG}(\text{DECIDE}, \perp, commitQC)$
As Replica:
Wait for message m from leader$(curView)$ matching QC COMMIT.
Execute new commands through $m.justify.node$.
Respond to clients.

▷ **View Interrupt**
On NextView Interrupt: (Triggered if $\text{NEXTVIEW}(curView)$ is called during any wait)
Send $\text{MSG}(\text{NEW-VIEW}, \perp, prepareQC)$ to leader$(curView + 1)$.
end for