On the Termination of the HotStuff Protocol Within the Universally Composable Framework

Yuhang Zeng, ZhiXin Dong, and Xian Xu
East China University of Science and Technology, Shanghai, China
y80240060@mail.ecust.edu.cn, dongzhixin@mail.ecust.edu.cn, xuxian@ecust.edu.cn

***Abstract*—HotStuff has gained widespread application in scenarios such as consortium chains in recent years due to its linear view change and pipelined decision making mechanisms. Although there have been studies on the performance of this algorithm, there remains a lack of analysis and formal termination proofs regarding its composability. This paper, for the first time, constructs a comprehensive formal system for the HotStuff protocol in a partially synchronous network environment under the Universally Composable (UC) framework and proves the termination of the HotStuff protocol within the UC framework. Specifically, we establish the ideal function and demonstrate that the HotStuff protocol UC realizes this ideal function, which guarantees the indistinguishability between the real protocol of HotStuff and the ideal function. Notably, in the context of network delay attacks, this paper uses a phased time analysis method with an exponential backoff timeout mechanism to show that the protocol can still achieve consensus within a bounded amount of time. The results of this work not only establish a formal proof paradigm for analyzing termination of BFT protocols in a composable framework, but also provide important theoretical foundations for ensuring reliable termination in industrial-grade blockchain systems (such as the Meta Diem project and Chang'an Chain that employ HotStuff).**

***Index Terms*—HotStuff, UC security framework, Network delay attack, Termination**

I. Introduction

In 1982, Lamport et al. [1] proposed the Byzantine Generals Problem, which models how nodes attempt to reach consensus despite the presence of malicious participants. The main difficulty lies in preventing faulty nodes from blocking progress. Byzantine Fault Tolerance (BFT) protocols are designed to ensure that, under certain network assumption (partially synchronize [2]), every valid client request is eventually processed and finalized within finite time. This termination property is crucial, as it guarantees that the system can continue to make progress despite adversarial interference, rather than being stalled indefinitely.

With the rise of consortium [3]and enterprise blockchains [4], consensus protocols face stricter performance demands under high concurrency and large scale nodes. BFT protocols are widely adopted to enhance fault tolerance and efficiency, but early schemes like PBFT [5] show clear limitations: its three phase protocol incurs $O(n^2)$ communication overhead, reducing throughput as nodes grow, and its leader based view change causes long recovery delays. These bottlenecks hinder PBFT from meeting modern blockchain requirements for high throughput, low latency, and scalability.

To overcome the limits of early BFT, new protocols such as HotStuff [6] and DumboBFT [7] adopt pipeline processing, message driven mechanisms, and threshold signatures, achieving throughput of tens of thousands of TPS and latency of only seconds. In addition, considering that distributed systems may encounter various failures in real world deployments, such as hardware failures [8], software errors [9], network problems [10], and malicious attacks, the importance of BFT protocols in ensuring system robustness has become increasingly prominent, and they have been widely used in key industries such as finance, supply chain, and government affairs.

HotStuff, in particular, enhances termination through Linear View Change and Pipelined Three Phase Commit, reducing communication complexity from $O(n^2)$ to $O(n)$. Deployed in projects like Facebook's Libra [11] (later Diem), it sustained 1,000+ TPS with sub-3s latency across 100 global nodes. However, its termination guarantees are largely analyzed under idealized assumptions (sequential execution, stable networks, rational participants). In real deployments with concurrency and adversarial behaviors (e.g., message reordering, state contamination), ensuring robust termination still requires further validation.

Canetti proposed the Universally Composable (UC) framework in 2001 [12] to ensure protocol security under arbitrary environments. Its core idea is to abstract target behavior into an ideal functionality and require real and ideal executions to be indistinguishable in any environment. The UC framework's main strength lies in composability and concurrent security: once a protocol realizes its ideal functionality, its security holds in arbitrary and concurrent settings. Owing to these properties, the UC model has become a standard tool for cryptographic protocol analysis, widely applied in blockchain consensus and BFT protocols to address challenges of concurrency and multi party interaction.

This paper, for the first time, modularly models HotStuff within the UC framework, formalizing its protocol logic. By introducing appropriate ideal functionalities and simulators into the UC model, we demonstrate that HotStuff achieves the expected termination guarantees under the specified adversarial model. This provides a rigorous foundation for reasoning about the protocol's ability to eventually decide, and offers a reusable paradigm for verifying termination of complex BFT protocols.

A. Related Work

1) Development of BFT Protocols: Since Lamport et al. introduced PBFT (Practical Byzantine Fault Tolerance), researchers have developed a spectrum of Byzantine Fault Tolerance (BFT) protocols focusing on termination guarantees under diverse network and fault models. Representative protocols such as PBFT, HoneyBadgerBFT [13], Tendermint [14], Algorand [15], and HotStuff explore different synchronization assumptions, leader election schemes, and message propagation methods to balance performance (e.g., throughput, latency, communication complexity) with security guarantees (e.g., fault tolerance, consistency).

PBFT employs a synchronous model with fixed timeouts and incurs $O(n^2)$ communication during view switches, limiting scalability. HoneyBadgerBFT ensures termination in asynchronous settings but suffers from high latency due to encrypted batching and random scheduling. Tendermint integrates blockchain with BFT voting, offering deterministic termination but risking stalls under leader failure. Algorand leverages VRF based sortition to reduce overhead, yet its probabilistic guarantees remain sensitive to network synchronization.

HotStuff addresses these limitations through linear view changes, pipelined decisions, and threshold signatures [16], reducing the complexity from $O(n^2)$ to $O(n)$ and improving throughput by 30–50%. Its modular design and efficiency gains have made it a practical consensus choice, applied in projects such as Facebook's Libra and vehicular networks.

2) Termination analysis of BFT Protocols: BFT protocols guarantee termination (i.e., eventual progress) through mechanisms such as view switching, random leader election, or asynchronous broadcast, while ensuring consistency and irreversibility through state machine replication and voting. As protocols grow more complex and deployment environments more diverse, formally proving termination has become a central challenge. Early works primarily offered partially formal, event-driven analyses: Castro and Liskov provided an informal yet foundational demonstration of PBFT's three-phase commit and view-switching mechanisms [17], while Veronese et al. [18] and Abd-El-Malek et al. [19] examined protocol behavior under different network models, giving preliminary assurances of eventual progress and practical deployability.

Formal tools like TLA+, Coq, and Ivy have been used to rigorously model and verify BFT protocols. Jehl formally specified HotStuff's linear view change and three-phase voting in Ivy/TLA+, proving consistency and termination in partially synchronous networks [20], supporting practical deployments such as LibraBFT. Recent works include Qiu et al.'s LiDO model for mechanized termination proofs of pipeline SMR protocols [21], Duan et al.'s dynamic BFT framework ensuring activity under membership changes [22], and Saltini and Hyland Wood's IBFT 2.0 demonstrating robust termination with dynamic validators [23]. Together, these establish formal foundations for termination and optimization of blockchain BFT protocols.

3) BFT Protocol and Universal Composable Framework: In a partially synchronous network [2], termination requires that after the Global Stabilization time (GST), client transactions are eventually processed and the protocol does not stall indefinitely due to Byzantine behavior. In other words, termination requires that the protocol must be able to terminate after the network is synchronized again, and the protocol process is not blocked indefinitely by Byzantine behavior.Most BFT protocols analyze termination in a "standalone" model—single-session, idealized networks, and isolated execution—which fails to capture concurrent sessions, inter-protocol interactions, or dynamic adversaries. In particular, latency attacks, such as delayed leader proposals or replica votes, can compromise termination. Existing analyses lack formal composability proofs, leaving uncertainty about whether termination holds under high concurrency, interleaved protocol executions, or adaptive network attacks, as seen in practical HotStuff deployments.

To overcome the limitations of "standalone" analysis, this paper adopts the Universally Composable (UC) framework, which formalizes protocol termination by proving that real world executions achieve the same progress guarantees as their ideal functionalities. Its core advantage is composability: subprotocols with proven termination retain this property when composed or invoked multiple times in any environment. The UC framework is widely applied in areas like digital signatures, key exchange, and broadcast authentication. Modeling activity requires showing that the protocol reliably produces outputs in any environment, despite concurrency or adversarial behavior. In the partially synchronous network model, messages are initially unpredictable but are bounded after global stabilization time (GST) by a constant Δ. By defining the ideal functionality $\mathcal{F}_{\mathrm{HotStuff}}$ and constructing a simulator $\mathcal{S}$, any environment $\mathcal{Z}$ cannot distinguish real from ideal execution. Termination is ensured implicitly: once the network is synchronized and inputs are valid, the real protocol must produce outputs in polynomial time, preventing $\mathcal{Z}$ from detecting differences via timeouts.

Thus, the definition of termination within the UC framework does not simply require a "final output." Instead, it requires that, once the network is synchronized and the environment provides valid input, the protocol always responds and outputs in polynomial time. This behavior can be naturally simulated in an ideal world by $\mathcal{F}_{\mathrm{HotStuff}}$ and the simulator $\mathcal{S}$. This modeling approach enables us to defend against highly concurrent and sophisticated attackers, ensuring that protocol termination is strictly guaranteed in terms of combinatorial safety.

Several works have applied the UC framework to BFT consensus. Gai et al. modeled a sidechain function for Cumulus but lacked a rigorous proof environment $\mathcal{Z}$, leaving real/ideal indistinguishability unresolved [24]. Cohen et al. defined UC functionalities for an asynchronous Byzantine protocol and showed composability in MPC, but without a full UC proof of termination [25]. In contrast, this paper constructs the ideal functionality $\mathcal{F}_{\mathrm{HotStuff}}$ and provides a rigorous UC termination proof in the partially synchronous

model, which to our knowledge is the first such formalization for HotStuff. These results establish strong UC guarantees for secure deployment in multi session, high concurrency settings.

B. Contribution

This paper analyzes the termination assurance of the HotStuff protocol within the UC framework. Specifically,

1) We construct a complete termination analysis model that modularly incorporates key components of HotStuff, including view switching logic ($\mathcal{F}_{\mathrm{nextview}}$), voting aggregation ($\mathcal{F}_{\mathrm{vote}}$), and quorum certificate generation ($\mathcal{F}_{\mathrm{QC}}$). We precisely characterize HotStuff's execution process, covering threshold signatures, the three stage voting procedure, view switching strategy, and new-view propagation;
2) We rigorously define the interaction interfaces and behavioral constraints of the environment, leader, replicas, and adversary, forming a full UC model. Also, we introduce an adversarial model where delayed messages trigger view switching failures, thereby capturing realistic network layer attacks on protocol termination;
3) We formalize both the ideal functionality $\mathcal{F}_{\mathrm{HotStuff}}$ and the real protocol π_{hotstuff}, proving that under the assumption $f < \frac{n}{3}$ and bounded network delay $\Delta < \Delta_{\max}$, π_{hotstuff} securely realizes $\mathcal{F}_{\mathrm{HotStuff}}$ in the UC sense. Altogether, these results establish that HotStuff achieves termination in the UC framework, thus safeguarding its compositonality in deployment.

Toward BFT protocols, this work expands the applicable boundaries of the UC security framework in termination analysis, and proposes an analysis process for modular and composable modeling.

Moreover, it provides a reusable technical framework and paradigm for the subsequent formal analysis of complex BFT-like consensus protocols in concurrent environments, which will hopefully help promote the use of the UC framework in the study of blockchain protocols.

C. Organization

The remaining sections are organized as follows. Section 2 introduces core mechanisms and optimizations of HotStuff, and the basics of the UC security framework. Section 3 constructs necessary ideal UC functionalities, and modeling network delay attacks, completing the ideal world modeling of the HotStuff within the UC framework. Section 4 formalizes the real world protocol π_{hotstuff}.Section 5 formally proves the termination of HotStuff under network latency attacks, and provides a global time complexity bound. Section 6 concludes, discusses limitations, and outlines future work.

II. Preliminary

A. HotStuff Consensus

HotStuff optimizes Byzantine consensus by reducing view change communication to linear $O(n)$. It achieves this through threshold signatures, which compress multiple votes into a constant size aggregate, and Quorum Certificates (QCs), which certify majority support and drive state advancement. These mechanisms cut broadcast overhead and enable consensus that is both scalable and high throughput while preserving termination.

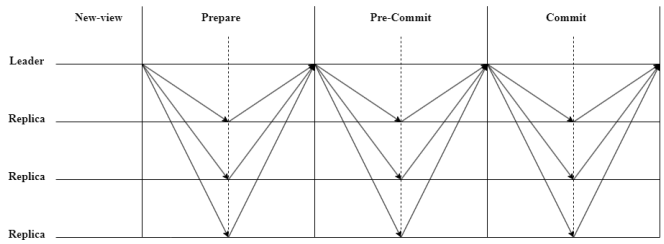

Fig. 1: The five phases of HotStuff consensus.

As shown in Fig. 1,The protocol operates through a New-view phase followed by a three-phase commit (Prepare, Pre-Commit, and Commit). In New-view, replicas send NEW-VIEW messages with their highest PrepareQC to the new leader, who selects the highest-QC as a safety anchor and proposes a new block. During each subsequent phase, replicas validate the proposal and send votes; the leader aggregates $2f+1$ signature shards to form a QC and broadcasts it. This sequential QC confirmation ensures safety and state continuity. The design achieves linear communication complexity ($O(n)$) by avoiding peer-to-peer broadcasts, and supports high throughput and modularity, making it suitable for dynamic, high load environments.

B. Universally Composable Model

The UC (Universally Composable) framework, proposed by Canetti [12] in 2001, enables modular reasoning about protocol properties, allowing complex systems to be constructed from simpler components. Within this framework, once a protocol's termination guarantees are proven, these assurances hold even when the protocol is executed concurrently or composed with others, avoiding the need to re-establish termination proofs for new scenarios and significantly reducing work effort.

1) UC Model Basics: The UC framework models all entities as Interactive Turing Machines (ITMs), each instantiated as an ITI identified by $(\mathrm{SID}, \mathrm{PID}, \mathrm{CodeID})$. By comparing real and ideal executions and enforcing unique pairs $(\mathrm{SID}, \mathrm{PID})$, it ensures protocol security and correctness under composition and concurrency.

The definition of UC security relies on protocol emulation: if a protocol π can simulate an ideal protocol ϕ in any environment such that their executions are indistinguishable, then π UC emulates ϕ. In the real world, an adversary $\mathcal{A}$ can corrupt participants, intercept or alter messages, while the environment $\mathcal{Z}$ interacts with both and finally outputs the real execution result $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(k,z)$, where k is the security parameter and z the input. In the ideal world, honest parties are replaced by dummy parties, ITMs that only forward inputs

from $\mathcal{Z}$ to the ideal functionality $\mathcal{F}$ and return their outputs. Since the dummy parties have no state, all logic reside in $\mathcal{F}$, and the adversary can only interact via $\mathcal{F}$, which simplifies simulation analysis.

In the UC framework, the ideal-world adversary is modeled by a simulator $\mathcal{S}$, which replaces the real-world adversary $\mathcal{A}$ by controlling corrupted parties and interacting with the ideal functionality $\mathcal{F}$ and the environment $\mathcal{Z}$. The simulator's task is to reproduce the externally observable effects of $\mathcal{A}$—including inputs, outputs, communication patterns, and message ordering—within the constraints of $\mathcal{F}$, which specifies the allowed leakage and delay semantics. The ideal execution is denoted by $\mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)$. A protocol π UC-realizes $\mathcal{F}$ if, for every real world adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(k,z)$ and $\mathsf{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)$ are computationally indistinguishable to any environment $\mathcal{Z}$.

$$\mathsf{EXEC}_{\pi,\mathrm{A},\mathrm{Z}}(k,z) \approx \mathsf{EXEC}_{\mathrm{F},\mathrm{S},\mathrm{Z}}(k,z)$$

A protocol π is said to UC securely implement an ideal functionality $\mathcal{F}$ if the above holds, i.e., for any PPT environment $\mathcal{Z}$, it cannot distinguish whether it interacts with the real world $(\pi,\mathcal{A})$ or the ideal world $(\mathcal{F},\mathcal{S})$. This definition guarantees security not only in isolation but also under arbitrary composition (e.g., as subprotocols, concurrent executions, or nested calls). Thus, UC security is universally composable: if π UC securely realizes $\mathcal{F}$, then any protocol ρ that uses $\mathcal{F}$ as a subroutine remains secure when $\mathcal{F}$ is replaced with π.

For instance, in blockchain systems, once the consensus protocol is proven UC terminating, higher-level protocols (e.g., smart contracts, payment or privacy mechanisms) can directly rely on its guaranteed progress without reanalyzing the consensus layer. This composability of termination is the UC framework's key advantage over standalone, simulation-based models.

2) The Generalized UC Framework: The generalized universal composability (GUC) framework [26] extends the basic UC model to handle protocols sharing a global setup, such as PKI, CRS, or public ledgers. Unlike the basic UC, where the environment $\mathcal{Z}$ interacts with a single protocol instance, GUC allows $\mathcal{Z}$ to concurrently run multiple sessions of π and any auxiliary protocols using the same global state.

To characterize such shared services, GUC introduces global ideal functions $\mathcal{F}^{\mathrm{g}}$, which exist independently of any individual session and are accessible to all participants and adversarial components at any time. In the formal definition, a protocol π is called a GUC simulation function $\mathcal{F}$ if for every probabilistic polynomial time (PPT) adversary $\mathcal{A}$ running in the real world, there exists a PPT simulator $\mathcal{S}$ in the ideal world, such that the unconstrained environment $\mathcal{Z}$ cannot distinguish between a real execution of the π protocol with $\mathcal{A}$ and an ideal functional execution of $\mathcal{F}$ with $\mathcal{S}$ (even if both share the same global functionality). Specifically, the indistinguishability requirement is expressed as follows:

$$\mathsf{GEXEC}_{\pi,\mathrm{A},\mathrm{Z}}(k,z) \approx \mathsf{GEXEC}_{\mathrm{F},\mathrm{S},\mathrm{Z}}(k,z)$$

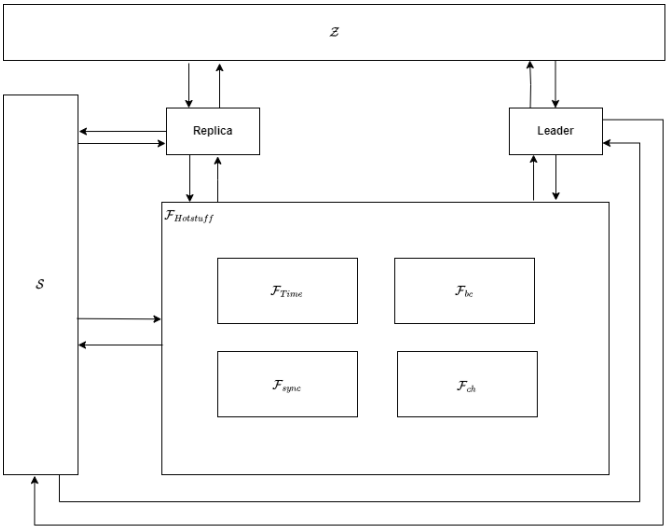

Fig. 2: Overall Framework of the HotStuff Protocol.

where $\mathcal{Z}$ is any PPT environment with auxiliary input z, and k a security parameter. $\mathsf{GEXEC}_{\pi,\mathcal{A},\mathcal{Z}}(k,z)$ and $\mathsf{GEXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k,z)$ denote the outputs of $\mathcal{Z}$ interacting with the real protocol π and adversary $\mathcal{A}$, or with the ideal functionality $\mathcal{F}$ and simulator $\mathcal{S}$, respectively. Computational indistinguishability means that no polynomial time environment can distinguish real from ideal execution with non-negligible probability. In the GUC framework, despite granting the environment additional power, the composability theorem holds: if a high level protocol ρ uses $\mathcal{F}$ as a subroutine and π GUC simulates $\mathcal{F}$, replacing $\mathcal{F}$ with π in ρ yields a protocol $\rho^{\mathcal{F}\to\pi}$ that still GUC simulates ρ

III. SYSTEM MODEL AND IDEAL FUNCTIONALITY

To analyze the security and liveness guarantees of the HotStuff protocol within the UC framework, we first construct a function in an ideal world, $\mathcal{F}_{\mathrm{HotStuff}}$, to abstractly describe all the core semantics and security properties that the protocol should implement. This ideal function, as an abstraction of real-world protocol behavior, forms the basis for the subsequent UC security proof, demonstrating that the HotStuff protocol UC implements this ideal function. Following the modeling paradigm of the UC framework, we divide the protocol functionality into several composable modules and formally define each of these modules, including communication behavior, security authentication, time advancement, and synchronization mechanisms.

We first model the idealized communication channel, including the point-to-point communication function $\mathcal{F}_{\mathrm{ch}}$, the public broadcast function $\mathcal{F}_{\mathrm{bc}}$, and the synchronization function $\mathcal{F}_{\mathrm{sync}}$ for ensuring phase and view synchronization. These modules provide message delivery guarantees for state propagation and round progression across HotStuff phases, forming

the fundamental communication structure for the multi-round consensus process.

Next, we model the authentication mechanisms used in the HotStuff protocol, including the identity authentication function $\mathcal{F}_{\text{auth}}$ and the threshold signature verification function $\mathcal{F}_{\text{sig}}$. These functions ensure the authenticity of proposal sources, the legitimacy of votes, and the verifiability of the resulting quorum certificate, forming the fundamental foundation for the protocol's security analysis.

Finally, we will move on to modeling the complete ideal world architecture, including the timing function $\mathcal{G}_{\text{time}}$ to simulate behaviors such as time window advancement and view timeout, and construct a comprehensive ideal function $\mathcal{F}_{\text{HotStuff}}$ to cover all core mechanisms such as leader rotation, phase transition, message collection and status update.

A. Communication Network Assumptions

1) Communication Channel: The enhanced channel functionality $\mathcal{F}_{\mathsf{Ch}}$ [27] supports flexible sender-receiver mappings with varying anonymity levels (see Fig. 3), including the sender-recipient anonymous channel $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{sra}}$, the sender-sender anonymous channel $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{ssa}}$, and the fully anonymous channel $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{fa}}$, all of which reveal only the message length $|m|$. By concealing both sender and receiver identities as well as message content, these channels mitigate timing correlation and traffic analysis under delay attacks. Their cryptographic guarantees ensure message integrity, while the use of gossip-based redundancy enhances robustness against partial message delays, thereby improving communication reliability in adversarial networks.

Communication Channel Functionality $\mathcal{F}_{\mathsf{Ch}}$

Let $\mathcal{P}$ define a set of parties where $\mathcal{S}$ and $\mathcal{R}$ denote two parties of the set as the sender and receiver of a message m respectively. Δ is defined as follows based on parameters of functionality. Message identifier mid is selected freshly by the functionality.

1) Upon input $\langle \mathsf{Send}, \mathsf{sid}, \mathcal{R}, m \rangle$ from $\mathcal{S}$, output $\langle \mathsf{Send}, \mathsf{sid}, \Delta, \mathsf{mid} \rangle$ to $\mathcal{A}$.
2) Upon receiving $\langle \mathsf{Ok}, \mathsf{sid}, \mathsf{mid} \rangle$ from $\mathcal{A}$, send $\langle \mathsf{Received}, \mathsf{sid}, \mathcal{S}, m \rangle$ to $\mathcal{R}$.

Set Δ based on the following parameterized functions:

- For $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{ac}}$ set $\Delta = (\mathcal{S}, \mathcal{R}, m)$. Upon receiving $\langle \mathsf{Ok.Snd}, \mathsf{sid}, \mathsf{mid} \rangle$ from $\mathcal{A}$, send $\langle \mathsf{Continue}, \mathsf{sid} \rangle$ to $\mathcal{S}$.[a]
- For $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{sra}}$ set $\Delta = (\mathcal{S}, |m|)$.
- For $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{ssa}}$ set $\Delta = (\mathcal{R}, |m|)$.
- For $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{fa}}$ set $\Delta = |m|$.
- For $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{sc}}$ set $\Delta = (\mathcal{S}, \mathcal{R}, |m|)$. Upon receiving $\langle \mathsf{Ok.Snd}, \mathsf{sid}, \mathsf{mid} \rangle$ from $\mathcal{A}$, send $\langle \mathsf{Continue}, \mathsf{sid} \rangle$ to $\mathcal{S}$.
- For $\mathcal{F}_{\mathsf{Ch}}^{\mathsf{sa}}$ set $\Delta = (\mathcal{R}, m)$.

Additional message handling rules:

1. Upon receiving $\langle \mathsf{Ok}, \mathsf{sid}, \mathsf{mid} \rangle$ from $\mathcal{A}$, send $\langle \mathsf{Received}, \mathsf{sid}, m, \mathsf{mid} \rangle$ to $\mathcal{R}$.
2. Upon receiving $\langle \mathsf{Send}, \mathsf{sid}, \mathsf{mid}, m' \rangle$ from $\mathcal{R}$, output $\langle \mathsf{Send}, \mathsf{sid}, \mathcal{R}, m', \mathsf{mid} \rangle$ to $\mathcal{A}$.

[a] This gives more power to adversary $\mathcal{A}$ who decides when the sender can proceed as sequential message sending is required in the UC model.

Fig. 3: Communication Channel Functionality $\mathcal{F}_{\mathsf{Ch}}$

Taking a completely covert channel ($\mathcal{F}_{\text{Ch}}^{\text{fa}}$) as an example: From the attacker's perspective, $\mathcal{A}$ can only observe that "a 256-byte message was transmitted at a certain time," but cannot determine who sent it, who received it, or what the content was. Even if the attacker $\mathcal{A}$ monitors network traffic for a long time, he or she will not be able to link user identities to behavioral patterns.

In the HotStuff protocol, $\mathcal{F}_{\text{ch}}$ is used for privacy-sensitive peer-to-peer communication (such as replicas voting for the leader). In the HotStuff consensus process, there are also multiple times when the leader broadcasts messages to all nodes. Therefore, we also need an ideal function $\mathcal{F}_{\text{bc}}$ that can achieve efficient global message synchronization to ensure the atomicity and timeliness of message delivery.

Broadcast Functionality $\mathcal{F}_{\mathsf{bc}}$

Broadcast functionality $\mathcal{F}_{\mathsf{bc}}$ parameterized by the set $\mathbb{M} = \{\mathsf{M}_1, \ldots, \mathsf{M}_D\}$ proceeds as follows:

- Upon receiving $\langle \mathsf{Broadcast}, \mathsf{sid}, m \rangle$ from a party P, send $\langle \mathsf{Broadcasted}, \mathsf{sid}, \mathsf{P}, m \rangle$ to all entities in the set $\mathbb{M}$ and to $\mathcal{A}$.

Fig. 4: Broadcast Functionality $\mathcal{F}_{\mathsf{bc}}$

2) Network synchronization: The UC framework is inherently asynchronous. To support the actual needs of high-concurrency consensus protocols such as HotStuff in the Changan Chain, we draw on the work of Dirk Achenbach et al. [21] and propose a synchronization function $\mathcal{F}_{\text{sync}}$ to achieve delay-bound synchronous network modeling. This mechanism requires each node to initiate a synchronization request to $\mathcal{F}_{\text{sync}}$ when entering a new consensus phase or switching views. The functional module tracks progress by maintaining node counters: when the counters of all nodes are updated, $\mathcal{F}_{\text{sync}}$ immediately broadcasts a synchronization signal and resets the counters. The synchronization barrier thus formed not only retains the asynchronous theoretical basis of the UC framework, but also provides timing guarantees for high-concurrency consensus in reality.

Functionality $\mathcal{F}_{\mathsf{sync}}$

Initialize for each party p_i a bit $d_i := 0$.

- Upon receiving message $\langle \mathsf{RoundOK} \rangle$ from party p_i set $d_i = 1$. If for all honest parties $d_i = 1$, then reset all d_i to 0. In any case, send $\langle \mathsf{switch}, \mathsf{p_i} \rangle$ to $\mathcal{A}$.
- Upon receiving message $\langle \mathsf{RequestRound} \rangle$ from p_i, send d_i to p_i.

Fig. 5: synchronization Functionality $\mathcal{F}_{\mathsf{sync}}$ [28]

B. Security authentication Function

1) Identity Verification: Considering that in the real world, attackers $\mathcal{A}$ may forge nodes to send false messages to interfere with the consensus process, we need to design an ideal function $\mathcal{F}_{\mathrm{auth}}$ to implement the registration and verification of honest nodes. Under the universal composability framework, the ideal function $\mathcal{F}_{\mathrm{auth}}$ provides anti-forgery attack protection for the distributed consensus protocol through a strict registration and dynamic verification mechanism. In this regard, we refer to the work of Canetti et al [29].

authentication Functionality $\mathcal{F}_{\mathsf{auth}}$

- Upon receiving $\langle \mathsf{Send}, \mathsf{sid}, B, m \rangle$ from party A, send $\langle \mathsf{Sent}, \mathsf{sid}, A, B, m \rangle$ to $\mathcal{A}$.
- Upon receiving $\langle \mathsf{Send}, \mathsf{sid}, B', m' \rangle$ from the $\mathcal{A}$, do: If A is corrupted: Output $\langle \mathsf{Sent}, \mathsf{sid}, A, m' \rangle$ to party B'. Else: Output $\langle \mathsf{Sent}, \mathsf{sid}, A, m \rangle$ to party B. Halt.
- Upon receiving $\langle \mathsf{Register}, \mathsf{sid}, A, pk \rangle$ from party A, send $\langle \mathsf{RegApply}, \mathsf{sid}, pk, A \rangle$ to $\mathcal{A}$.
- Upon receiving $\langle \mathsf{RegStatus}, \mathsf{sid}, A \rangle$ from the $\mathcal{A}$, do: If RegStatus == 1: Send $\langle \mathsf{RegisterSuccess}, \mathsf{sid}, A \rangle$ to party A. Else: Output $\langle \mathsf{RegisterFailure}, \mathsf{sid}, A \rangle$ to party A. Halt.
- Upon receiving $\langle \mathsf{Lookup}, \mathsf{sid}, A, pk \rangle$ from party A, send $\langle \mathsf{LookApply}, \mathsf{sid}, pk, A \rangle$ to $\mathcal{A}$.
- Upon receiving $\langle \mathsf{LookStatus}, \mathsf{sid}, A \rangle$ from the $\mathcal{A}$, do: If LookStatus == 1: Send $\langle \mathsf{LookupSuccess}, \mathsf{sid}, A \rangle$ to party A. Else: Output $\langle \mathsf{LookupFailure}, \mathsf{sid}, A \rangle$ to party A. Halt.
- Upon receiving $\langle \mathsf{Delete}, \mathsf{sid}, A, pk \rangle$ from party A, send $\langle \mathsf{DeleteApply}, \mathsf{sid}, pk, A \rangle$ to $\mathcal{A}$.
- Upon receiving $\langle \mathsf{DeleteStatus}, \mathsf{sid}, A \rangle$ from the $\mathcal{A}$, do: If DeleteStatus == 1: Send $\langle \mathsf{DeleteSuccess}, \mathsf{sid}, A \rangle$ to party A. Else: Output $\langle \mathsf{DeleteFailure}, \mathsf{sid}, A \rangle$ to party A. Halt.

Fig. 6: authentication Functionality $\mathcal{F}_{\mathsf{auth}}$

signature Functionality $\mathcal{F}_{\mathsf{sig}}$

Key Generation

- Upon receiving a value $\langle \mathsf{KeyGen}, \mathsf{sid} \rangle$ from some party S, verify that $\mathsf{sid} = \langle \mathsf{S}, \mathsf{sid}' \rangle$ for some sid'. If not, then ignore the request. Else, hand $\langle \mathsf{KeyGen}, \mathsf{sid} \rangle$ to the adversary.
- Upon receiving $\langle \mathsf{VerificationKey}, \mathsf{sid}, \mathsf{pid}, v \rangle$ from the adversary, output $\langle \mathsf{VerificationKey}, \mathsf{sid}, \mathsf{pid}, v \rangle$ to S, and record the pair $\langle \mathsf{S}, v \rangle$.

signature Generation

- Upon receiving a value $\langle \mathsf{sign}, \mathsf{sid}, \mathsf{pid}, m \rangle$ from S, verify that $\mathsf{sid} = \langle \mathsf{S}, \mathsf{sid}' \rangle$ for some sid'. If not, then ignore the request. Else, send $\langle \mathsf{sign}, \mathsf{sid}, \mathsf{pid}, m \rangle$ to the adversary.
- Upon receiving $\langle \mathsf{signature}, \mathsf{sid}, \mathsf{pid}, m, \sigma \rangle$ from the adversary, verify that no entry $\langle m, \sigma, 0 \rangle$ is recorded. If it is, then output an error message to S and halt. Else, output $\langle \mathsf{signature}, \mathsf{sid}, \mathsf{pid}, m, \sigma \rangle$ to S, and record the entry $\langle m, \sigma, 1 \rangle$.

signature Verification

- Upon receiving a value $\langle \mathsf{Verify}, \mathsf{sid}, \mathsf{pid}, m, \sigma, v' \rangle$ from party P, hand $\langle \mathsf{Verify}, \mathsf{sid}, \mathsf{pid}, m, \sigma, v' \rangle$ to the adversary. Upon receiving $\langle \mathsf{Verified}, \mathsf{sid}, \mathsf{pid}, m, \phi \rangle$ from the adversary, do:
  1) If $v' = v$ and the entry $\langle m, \sigma, 1 \rangle$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
  2) Else, if $v' = v$, the signer is not corrupted, and no entry $\langle m, \sigma', 1 \rangle$ for any σ' is recorded, then set $f = 0$. (This condition guarantees unforgeability: If v' is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
  3) Else, if there is an entry $\langle m, \sigma, f' \rangle$ recorded then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
  4) Else, $f = \phi$ and record the entry $\langle m, \sigma', \phi \rangle$.

  Output $\langle \mathsf{Verified}, \mathsf{id}, m, f \rangle$ to P.

Fig. 7: signature Functionality $\mathcal{F}_{\mathsf{sig}}$

During the system initialization phase, all replica nodes need to submit a registration request containing a public key to $\mathcal{F}_{\mathrm{auth}}$. The simulator S performs offline verification of the node identity. Only when $\mathcal{S}$ returns the authentication status $\mathrm{RegStatus} = 1$ can the node join the replica set and participate in the protocol execution, thereby eliminating unauthorized entity infiltration at the ideal world level.

During the protocol's operation, $\mathcal{F}_{\text{auth}}$ performs real-time verification on every received message: it initiates a certificate verification request $(\text{VerifyCert}, \text{sid}, \text{A}, \text{pk})$ to S and uses the $cert_j$ returned by the simulator $\mathcal{S}$ to determine the message's legitimacy. If verification fails, the corresponding node is immediately marked as malicious and removed from the replica set, simultaneously blocking any subsequent attack attempts.

This mechanism, through admission control during the registration phase and dynamic detection at runtime, establishes a trusted communication layer within the UC composability framework, preventing attackers $\mathcal{A}$ from forging valid identities or signatures. This provides a formal security foundation for the HotStuff protocol to defend against network partition attacks.

2) signature Verification: In addition to identity authentication, we also need to verify the signatures attached to all messages in the protocol process. Therefore, we modified the work of Canetti et al. [23] and designed a signature verification function $\mathcal{F}_{\text{sig}}$. $\mathcal{F}_{\text{sig}}$ mainly includes three modules: key management, signature generation and verification:

First, the key pair $(\text{pk}_\text{i}, \text{sk}_\text{i})$ of each replica node r_i is generated and hosted by $\mathcal{F}_{\text{sig}}$ through a secure channel during the initialization phase, ensuring that the private key sk_i is only visible to $\mathcal{S}$, thereby avoiding the risk of key leakage in the real world.

Second, when the node needs to sign the message m, $\mathcal{F}_{\text{sig}}$ receives the $(\text{sign}, , \text{sid}, , \text{m})$ request and forwards it to $\mathcal{S}$, who then generates a signature based on the escrowed private key and records it in the signature log to prevent replay attacks.

Finally, any recipient can submit a $(\text{Verify}, \text{sid}, \text{m}, \sigma, \text{v}')$ request to $\mathcal{F}_{\text{sig}}$, whereupon $\mathcal{S}$ verifies the binding between the signature and the message. If the signature is invalid or conflicts with the historical record, it is marked as forged and the malicious node isolation process is triggered.

This mechanism, through collaborative verification with $\mathcal{F}_{\text{auth}}$ (double verification of certificate authentication and signature verification), ensures that an attacker $\mathcal{A}$ cannot forge a legitimate identity, tamper with, or replay signed messages.

C. The timer Functionality $\mathcal{G}_{\mathsf{time}}$

To ensure protocol termination under network delay attacks, the HotStuff protocol integrates a timeout mechanism formalized as the ideal functionality $\mathcal{G}_{\mathsf{time}}$ within the UC framework. This functionality models a logical timer enabling honest parties to detect and respond to extended inactivity across protocol phases, serving as a fundamental building block for progress guarantees under partial synchrony, where message delays are eventually bounded. $\mathcal{G}_{\mathsf{time}}$ maintains phase-specific timers for protocol sessions identified by unique session identifiers sid, initially set to $\perp$. Honest parties interact via three interfaces: Gettime to query the current timer value, Resettime to clear it, and $\langle \mathsf{timeStart}, \mathsf{sid}, \mathsf{phase_j}, \delta \rangle$ to initiate a countdown of δ time units. In compliance with UC scheduling semantics and upon expiration, the invoking party obtains a timeout notification $\langle \mathsf{timeOver}, \mathsf{sid}, \mathsf{phase_j}, \delta \rangle$. This irrevocable countdown—immune to adversarial preemption—ensures deterministic timeout behavior crucial for HotStuff's resilience: if proposals or quorum votes fail to arrive within the timeout, the protocol deterministically advances to the next round, visible to honest parties but hidden from the adversary. The formal specification is provided (see Fig. 8), and $\mathcal{G}_{\mathsf{time}}$ satisfies UC simulatability requirements, underpinning HotStuff's termination proof under universal composition.

timer Functionality $\mathcal{G}_{\mathsf{time}}$

The functionality $\mathcal{G}_{\mathsf{time}}$ models a logical countdown timer. It is parameterized over a global discrete time domain T, and maintains an internal table of timers indexed by session identifiers sid. Each entry t_j either stores a countdown integer or $\perp$. For all j, initialize $t_\text{j} := \perp$.

- Upon receiving $\langle \mathsf{Gettime}, \mathsf{sid} \rangle$ from a replica r_j or the environment, return the current value of t_j to replica r_j.
- Upon receiving $\langle \mathsf{Resettime}, \text{j} \rangle$ from a replica r_j, set $t_\text{j} := \perp$ and return $\langle \mathsf{timeOK}, \text{j} \rangle$ to replica r_j.
- Upon receiving $\langle \mathsf{timeStart}, \mathsf{sid}, \mathsf{phase}_j, \delta \rangle$ from a replica r_j: if $t_\text{j} \neq \perp$, ignore the request.
- Otherwise:
  1) Set $t_\text{j} := \delta$, and return $\langle \mathsf{timeOK}, \text{j} \rangle$ to replica r_j.
  2) From this point onward, in each global round, decrement t_j by 1 if $t_\text{j} \in \mathbb{N}$.
  3) When $t_\text{j} = 0$, send $\langle \mathsf{timeOver}, \text{j}, \mathsf{phase}_j, \delta \rangle$ to replica r_j.

Fig. 8: timer Functionality $\mathcal{G}_{\mathsf{time}}$

D. Ideal World Modeling

The Functionality $\mathcal{F}_{\mathsf{HotStuff}}$

Corrupt

Upon receiving a message $\langle \mathsf{Corrupt}, \mathsf{sid}, \text{j}, \mathsf{Replica} \rangle$:

- If $\mathsf{Replica}_j \in \{Replica_1, ..., Replica_n\}$, then record $Replica_j$ as corrupted.

NewView

Upon receiving message $MSG\langle \text{NEW-VIEW}, ViewNumber \rangle$ from Leader_i: Send $\langle \text{VerifyCert}, \text{sid}, \text{Leader}_i, cert_i \rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle \text{CertValid}, \text{sid}, \text{Leader}_i, 1/0 \rangle$:

- If receive $\langle \text{CertValid}, \text{sid}, \text{Leader}_i, 1 \rangle$ from $\mathcal{F}_{auth}$:,
  - $\text{curView}_i := \text{ViewNumber} + 1$
- If Leader_i is corrupted:
  - Send $\langle \text{Input}, \text{curView}_i, \text{cmd} \rangle$ to $\mathcal{S}$.
- Send$\langle \text{Prepare}, \text{i}, \text{curView}_i, \text{sid} \rangle$ to $\mathcal{S}$ wait for a re-

sponse of the form $\langle$startPrepare, i, curView$_i$, sid$\rangle$:

- Set phase$_j$:= Prepare which curView$_j$ = curView$_i$.

- Else: Remove Leader$_i$ from the Replica Set.
  - Send$\langle$NEW-VIEW, curView$_i$, $i\rangle$ to $\mathcal{S}$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
    - If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
    - Else re-execute this step.

Prepare

Upon receiving message$MSG\langle$NEW-VIEW, $\perp$, PrepareQC$\rangle$ from Replica r$_j$:

Send $\langle$Sleep, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$Wake, sid, phase$_j$, δ, $\sigma^{L_i}_{prepare}\rangle$:

- If ($\delta + \sigma^{L_i}_{prepare} > \Delta^i_{vnum}$) $\vee$ $\sigma^{L_i}_{prepare} > \sigma$:
  - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$
  - send$\langle$NEW-VIEW, curView$_i$, $i\rangle$ to $\mathcal{S}$ and Leader$_i$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
  - If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
  - else re-execute this step.
- Send $\langle$Verify, sid, m, sig$_j$, pk$_j\rangle$ to $\mathcal{F}_{\text{sig}}$, wait for response $\langle$Verified, sid, m, sig$_j\rangle$.
- Send $\langle$Verify, sid, r$_j$, pk$_j\rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle$Verified, sid, r$_j$, cert$_j\rangle$.
- If $sig_j = 0 \vee cert_j = 0$:
  - Ignore the message
- If MATCHINGMSG$\langle$m, Prepare, curView$_j\rangle$ $\cap$ step$_j$ = Prepare:
  - Set nums$_i$(NEW-VIEW) := nums$_i$(NEW-VIEW) + 1.
  - If (nums$_i$(NEW-VIEW) $> 2f + 1$),Send $\langle$CreateProposal, i, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$StartProposal, i, sid$\rangle$:
    - Create$\langle$Prepare, Curproposal, highQC$\rangle$.
    - If no$\langle$timeOver, sid$\rangle$ is received from $\mathcal{G}_{time}$:
      - send $\langle$Prepare, Curproposal, highQC$\rangle$ to $\mathcal{F}_{bc}$.
      - send $\langle$updatePreCommit, i, sid$\rangle$ to $\mathcal{S}$,wait for the response $\langle$startPreCommit,j,sid$\rangle$.
      - Set phase$_i$ = PreCommit
    - Else:
      - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$.
      - send$\langle$NEW-VIEW, curView$_i$, $i\rangle$ to $\mathcal{S}$ and Leader$_i$.
      - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
      - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
      - If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
      - else re-execute this step.
- Else,ignore this message.

Upon receiving message $MSG\langle$Prepare, CurProposal, $highQC\rangle$ from Leader$_i$: Send $\langle$Sleep, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form$\langle$Wake, sid, phase$_j$, δ, $\sigma^{R_j}_{prepare}\rangle$:

- If ($\delta + \sigma^{R_j}_{prepare} > \Delta^j_{vnum}$) $\vee$ $\sigma^{R_j}_{prepare} > \sigma$:
  - set $\Delta^j_{vnum} := \Delta^j_{vnum} * 2$, curView$_j$:= curView$_j + 1$
  - send$\langle$NEW-VIEW, curView$_j$, $j\rangle$ to $\mathcal{S}$ and Leader$_i$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
  - If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
  - else re-execute this step.
- Send $\langle$Verify, sid, m, sig$_i$, pk$_i\rangle$ to $\mathcal{F}_{\text{sig}}$, wait for response $\langle$Verified, sid, m, sig$_i\rangle$.
- Send $\langle$Verify, sid, r$_i$, pk$_i\rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle$Verified, sid, r$_i$, cert$_i\rangle$.
- If $sig_i = 0 \vee cert_i = 0$:
  - Ignore the message
- If MATCHINGMSG$\langle$m, Prepare, curView$_j\rangle$ $\cap$ phase$_j$ = Prepare:
  - If the highQC is higher than PrepareQC$_j$, update PrepareQC$_j \leftarrow$ highQC, LockedQC$_j \leftarrow$ CommitQC$_i$
  - If SafeNode$(m.node, m.justify)$ $\cap$ m.node extends from m.justify.node and no $\langle$timeOver, sid$\rangle$ has been received from $\mathcal{G}_{time}$:
    - Send $\langle$Prepare, $m.node, \perp\rangle$ to Leader$_i$.
    - Send $\langle$updatePreCommit, j, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$StartPreCommit, j, sid$\rangle$:
    - Set phase$_j$ = PreCommit.
  - Else:
    - set $\Delta^j_{vnum} := \Delta^j_{vnum} * 2$
    - send$\langle$NEW-VIEW, curView$_j$, $j\rangle$ to $\mathcal{S}$ and Leader$_i$.
    - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
    - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_j.
    - If $d_j = 0$,set curView$_j$:= curView$_j + 1$.
    - else re-execute this step.

PreCommit

Upon receiving message$VOTEMSG\langle$Prepare, $m.node, \perp$

$\rangle$ from Replica r_j:
Send $\langle$Sleep, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$Wake, sid, phase$_j, \delta, \sigma^{L_i}_{precommit}\rangle$:

- If ($\delta + \sigma^{L_i}_{prepare} + \sigma^{L_i}_{precommit} > \Delta^i_{vnum}$) $\vee$ $\sigma^{L_i}_{precommit} > \sigma$:
  - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$
  - send$\langle$NEW-VIEW, curView$_i, i\rangle$ to $\mathcal{S}$ and Leader$_i$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
  - If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
  - else re-execute this step.
- Send $\langle$Verify, sid, $m, \mathsf{sig_j}, \mathsf{pk_j}\rangle$ to $\mathcal{F}_{\mathsf{sig}}$, wait for response $\langle$Verified, sid, m, sig$_j\rangle$.
- Send $\langle$Verify, sid, $\mathsf{r}_j, \mathsf{pk}_j\rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle$Verified, sid, $\mathsf{r}_j, \mathsf{cert}_j\rangle$.
- If $sig_j = 0 \vee cert_j = 0$:
  - Ignore the message
- Else if MATCHINGMSG$\langle$m, Prepare, curView$_i\rangle \cap$ phase$_i$ = PreCommit:
  - If (nums$_i(Prepare) > 2f + 1$): Create$\langle$PreCommit, $\perp$, PrepareQC$\rangle$.
  - If no$\langle$timeOver, sid$\rangle$ is received from $\mathcal{G}_{time}$:
    - Send $\langle$PreCommit, $\perp$, PrepareQC$\rangle$to $\mathcal{F}_{bc}$.
    - Send $\langle$updateCommit, i, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$StartCommit, j, sid$\rangle$:
    - Set phase$_i$ = Commit.
  - Else:
    - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$
    - send$\langle$NEW-VIEW, curView$_i, i\rangle$ to $\mathcal{S}$ and Leader$_i$.
    - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
    - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
    - If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
    - else re-execute this step.

  Upon receiving message $MSG\langle$PreCommit, $\perp$, PrepareQC$\rangle$ from Leader$_i$: Send $\langle$Sleep, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form$\langle$Wake, sid, phase$_j, \delta, \sigma^{R_j}_{precommit}\rangle$:
- If ($\delta + \sigma^{R_j}_{prepare} + \sigma^{R_j}_{precommit} > \Delta^j_{vnum}$) $\vee$ $\sigma^{R_j}_{precommit} > \sigma$:
  - set $\Delta^j_{vnum} := \Delta^j_{vnum} * 2$
  - send$\langle$NEW-VIEW, curView$_j, j\rangle$ to $\mathcal{S}$ and Leader$_i$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_j.
  - If $d_j = 0$,set curView$_j$:= curView$_j + 1$.
  - else re-execute this step.
- Send $\langle$Verify, sid, m, sig$_\mathsf{i}$, pk$_\mathsf{i}\rangle$ to $\mathcal{F}_{\mathsf{sig}}$, wait for response $\langle$Verified, sid, m, sig$_i\rangle$.
- Send $\langle$Verify, sid, $\mathsf{r}_i, \mathsf{pk}_i\rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle$Verified, sid, $\mathsf{r}_i, \mathsf{cert}_i\rangle$.
- If $sig_i = 0 \vee cert_i = 0$:
  - Ignore the message
- Else if MATCHINGQC$\langle$m.justify, Prepare, curView$_j\rangle$ $\cap$ phase$_j$ = PreCommit:
  - If no $\langle$timeOver, sid$\rangle$ has been received from $\mathcal{G}_{time}$:
    - Set PrepareQC$_j \leftarrow$ m.justify.
    - Send VOTEMSG$\langle$PreCommit, m.justify.node , $\perp\rangle$ to Leader$_i$.
    - Send $\langle$updateCommit, j, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$StartCommit, j, sid$\rangle$:
    - Set phase$_j$ = Commit.
  - Else:
    - set $\Delta^j_{vnum} := \Delta^j_{vnum} * 2$
    - send$\langle$NEW-VIEW, curView$_j, j\rangle$ to $\mathcal{S}$ and Leader$_i$.
    - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
    - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_j.
    - If $d_j = 0$,set curView$_j$:= curView$_j + 1$.
    - else re-execute this step.

Commit
Upon receiving message $VOTEMSG\langle$PreCommit, $m.node, \perp\rangle$ from Replica r_j:
Send $\langle$Sleep, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$Wake, sid, phase$_j, \delta, \sigma^{L_i}_{commit}\rangle$:

- If ($\delta + \sigma^{L_i}_{prepare} + \sigma^{L_i}_{precommit} + \sigma^{L_i}_{commit} > \Delta^i_{vnum}$) $\vee$ $\sigma^{L_i}_{commit} > \sigma$:
  - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$
  - send$\langle$NEW-VIEW, curView$_i, i\rangle$ to $\mathcal{S}$ and Leader$_i$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
  - If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
  - else re-execute this step.
- Send $\langle$Verify, sid, $m, \mathsf{sig_j}, \mathsf{pk_j}\rangle$ to $\mathcal{F}_{\mathsf{sig}}$, wait for response $\langle$Verified, sid, m, sig$_j\rangle$.
- Send $\langle$Verify, sid, $\mathsf{r}_j, \mathsf{pk}_j\rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle$Verified, sid, $\mathsf{r}_j, \mathsf{cert}_j\rangle$.
- If $sig_j = 0 \vee cert_j = 0$:
  - Ignore the message

- Else if MATCHINGMSG$\langle$m, PreCommit, curView$_i\rangle\cap$ phase$_i$ = Commit:
  - Set nums$_i$(PreCommit) := nums$_i$(PreCommit) + 1
  - If (nums$_i$(PreCommit) $>$ $2f$ + 1):Create$\langle$Commit, $\perp$, PreCommitQC$\rangle$.
  - If no$\langle$timeOver, sid$\rangle$ is received from $\mathcal{G}_{time}$:
    - Send $\langle$Commit, $\perp$, PreCommitQC$\rangle$to $\mathcal{F}_{bc}$.
    - Send $\langle$updateDecide, i, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$StartDecide, j, sid$\rangle$:
    - Set phase$_i$ = Decide.
  - Else:
    - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$
    - send$\langle$NEW-VIEW, curView$_i, i\rangle$ to $\mathcal{S}$ and Leader$_i$.
    - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
    - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
    - If $d_i = 0$,set curView$_i$:= curView$_i$ + 1.
    - else re-execute this step.

  Upon receiving message $MSG\langle$Commit, $\perp$, PreCommitQC$\rangle$ from Leader$_i$:
  Send $\langle$Sleep, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$Wake, sid, phase$_j$, δ, $\sigma^{R_j}_{commit}\rangle$:
- If ($\delta + \sigma^{R_j}_{prepare} + \sigma^{R_j}_{precommit} + \sigma^{R_j}_{commit} > \Delta^j_{vnum}$) $\vee$ $\sigma^{R_j}_{commit} > \sigma$:
  - Set $\Delta^j_{vnum} := \Delta^j_{vnum} * 2$
  - Send$\langle$NEW-VIEW, curView$_j, j\rangle$ to $\mathcal{S}$ and Leader$_i$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_j.
  - If $d_j = 0$,set curView$_j$:= curView$_j$ + 1.
  - else re-execute this step.
- Send $\langle$Verify, sid, m, sig$_i$, pk$_i\rangle$ to $\mathcal{F}_{\mathsf{sig}}$, wait for response $\langle$Verified, sid, m, sig$_i\rangle$.
- Send $\langle$Verify, sid, r$_i$, pk$_i\rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle$Verified, sid, r$_i$, cert$_i\rangle$.
- If $sig_i = 0 \vee cert_i = 0$:
  - Ignore the message
- Else if MATCHINGQC $\langle$m.justify, PreCommit, curView$_j\rangle$ $\cap$ phase$_j$ = Commit:
  - If no $\langle$timeOver, sid$\rangle$ has been received from $\mathcal{F}_{time}$:
    - Set LockedQC$_j \leftarrow$ m.justify.
    - Send VOTEMSG$\langle$Commit, m.justify.node, $\perp$ $\rangle$ to Leader$_i$.
    - Send $\langle$updateDecide, j, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form $\langle$StartDecide, j, sid$\rangle$:
    - Set phase$_j$ = Decide.
  - Else:
    - set $\Delta^j_{vnum} := \Delta^j_{vnum} * 2$
    - send$\langle$NEW-VIEW, curView$_j, j\rangle$ to $\mathcal{S}$ and Leader$_i$.
    - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
    - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_j.
    - If $d_j = 0$,set curView$_j$:= curView$_j$ + 1.
    - else re-execute this step.

Decide

Upon receiving message $VOTEMSG\langle$Commit, $m.node$, $\perp\rangle$ from Replica r_j:
Send $\langle$Sleep, sid$\rangle$ to $\mathcal{S}$ and wait for a response of the form$\langle$Wake, sid, phase$_j$, δ, $\sigma^{L_i}_{decide}\rangle$:

- If ($\delta + \sigma^{L_i}_{prepare} + \sigma^{L_i}_{precommit} + \sigma^{L_i}_{commit} + \sigma^{L_i}_{decide} > \Delta^i_{vnum}$) $\vee$ $\sigma^{L_i}_{decide} > \sigma$:
  - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$
  - send$\langle$NEW-VIEW, curView$_i, i\rangle$ to $\mathcal{S}$ and Leader$_i$.
  - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
  - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
  - If $d_i = 0$,set curView$_i$:= curView$_i$ + 1.
  - else re-execute this step.
- Send $\langle$Verify, sid, m, sig$_j$, pk$_j\rangle$ to $\mathcal{F}_{\mathsf{sig}}$, wait for response $\langle$Verified, sid, m, sig$_j\rangle$.
- Send $\langle$Verify, sid, r$_j$, pk$_j\rangle$ to $\mathcal{F}_{auth}$, wait for response $\langle$Verified, sid, r$_j$, cert$_j\rangle$.
- If $sig_j = 0 \vee cert_j = 0$:
  - Ignore the message
- Else if MATCHINGMSG$\langle$m, Commit, curView$_i\rangle\cap$ phase$_i$ = Decide:
  - Set nums$_i$(Commit) := nums$_i$(Commit) + 1
  - If (nums$_i$(Commit) $> 2f + 1$):
  - If no$\langle$timeOver, sid$\rangle$ is received from $\mathcal{G}_{time}$:
    - send $\langle$Decide, $\perp$, CommitQC$\rangle$to $\mathcal{F}_{bc}$.
    - send$\langle$NEW-VIEW, curView$_i, i\rangle$ to $\mathcal{S}$ and Leader$_i$.
    - Send $\langle$RoundOK$\rangle$ to $\mathcal{F}_{sync}$.
    - Send $\langle$RequestRound$\rangle$ to $\mathcal{F}_{sync}$,receive its response d_i.
    - If $d_i = 0$,set curView$_i$:= curView$_i$ + 1.
    - else re-execute this step.
    - Else:
      - set $\Delta^i_{vnum} := \Delta^i_{vnum} * 2$
      - send$\langle$NEW-VIEW, curView$_i, i\rangle$ to $\mathcal{S}$ and Leader$_i$.

· Send ⟨RoundOK⟩ to $\mathcal{F}_{sync}$.
· Send ⟨RequestRound⟩ to $\mathcal{F}_{sync}$,receive its response d_i.
· If $d_i = 0$,set curView$_i$:= curView$_i + 1$.
· else re-execute this step.

Upon receiving message $MSG\langle$Decide, $\perp$, CommitQC$\rangle$ from Leader$_i$:
Send ⟨Sleep, sid⟩ to $\mathcal{S}$ and wait for a response of the form ⟨Wake, sid, phase$_j$, δ, $\sigma^{R_j}_{decide}\rangle$:

- If ($\delta + \sigma^{R_j}_{prepare} + \sigma^{R_j}_{precommit} + \sigma^{R}_{commit} + \sigma^{R}_{decide} > \Delta^{j}_{vnum}$) $\vee$ $\sigma^{R}_{decide} > \sigma$:
  - Set $\Delta^{j}_{vnum} := \Delta^{j}_{vnum} * 2$, curView$_j$:= curView$_j + 1$.
  - Send⟨NEW-VIEW, curView$_j, j\rangle$ to $\mathcal{S}$ and Leader$_i$.
- Send ⟨Verify, sid, m, sig$_i$, pk$_i$⟩ to $\mathcal{F}_{\mathsf{sig}}$, wait for response ⟨Verified, sid, m, sig$_i\rangle$.
- Send ⟨Verify, sid, r$_i$, pk$_i\rangle$ to $\mathcal{F}_{auth}$, wait for response ⟨Verified, sid, r$_i$, cert$_i\rangle$.
- If $sig_i = 0 \vee cert_i = 0$:
  - Ignore the message
- Else if MATCHINGQC⟨m.justify, Commit, curView$_j\rangle$ $\cap$ phase$_j$ = Decide:
  - If no ⟨timeOver, sid⟩ has been received from $\mathcal{G}_{time}$:
    * Execute new commands through m.justify.node
    * Send ⟨RoundOK⟩ to $\mathcal{F}_{sync}$.
    * Send ⟨RequestRound⟩ to $\mathcal{F}_{sync}$,receive its response d_j.
    * If $d_j = 0$,set curView$_j$:= curView$_j + 1$.
    * else re-execute this step.
  - Else:
    * set $\Delta^{j}_{vnum} := \Delta^{j}_{vnum} * 2$
    * send⟨NEW-VIEW, curView$_j, j\rangle$ to $\mathcal{S}$ and Leader$_i$.
    * Send ⟨RoundOK⟩ to $\mathcal{F}_{sync}$.
    * Send ⟨RequestRound⟩ to $\mathcal{F}_{sync}$,receive its response d_j.
    * If $d_j = 0$,set curView$_j$:= curView$_j + 1$.
    * else re-execute this step.
- If (count$_{\mathsf{nextround}} > f + 1) \wedge$ round $>$ round$_p$:
  - Send ⟨NEWROUND, h_p, round, $*\rangle$ to $\mathcal{S}$.

Fig. 9: The Functionality $\mathcal{F}_{\mathsf{HotStuff}}$

In the ideal world, core UC functionalities—including point-to-point channels $\mathcal{F}_{\mathrm{ch}}$, broadcast $\mathcal{F}_{\mathrm{bc}}$, synchronization $\mathcal{F}_{\mathrm{sync}}$, authentication $\mathcal{F}_{\mathrm{auth}}$, threshold signatures $\mathcal{F}_{\mathrm{sig}}$, and the clock $\mathcal{G}_{\mathrm{time}}$—model secure communication, identity verification, and timing control. As illustrated in Fig. 2, the environment $\mathcal{Z}$ drives protocol execution, the ideal functionality $\mathcal{F}_{\mathrm{HotStuff}}$ encapsulates consensus logic, and the simulator $\mathcal{S}$ maps adversarial actions into ideal-world perturbations, ensuring indistinguishable executions and reducing HotStuff's termination guarantees to the assumptions of the ideal functionalities.

TABLE I: Symbol Explanation of $\mathcal{F}_{\mathsf{HotStuff}}$

Symbol	Explanation
δ	Normal protocol execution time, provided by the configuration file.
σ	Adversary's attack delay, specified by $\mathcal{S}$.
phase$_j$	Current phase of node j (prepare, precommit, commit, decide).
Δ	Initial maximum timeout window.
Δ^{i}_{vnum}	Current phase of node j Maximum timeout window for replica r_i within the current view.
σ	Maximum delay in delay attack.
cert$_j$	identity's validity of replica r_j.
sig$_j$	signature's validity of replica r_j.
curView$_j$	Current view number of replica r_j.
m	Message or voting message.
m.node	Node that proposed the proposal in the current view.
m.justify	Quorum Certificate carried by the message
curView$_j$	Current view number of replica r_j.
curView$_j$	Current view number of replica r_j.
curView$_j$	Current view number of replica r_j.

Next, we define network delay attack in the context of HotStuff protocol consensus:

Definition 1 (Network Delay Attack)**.** An attacker can disrupt consensus efficiency by controlling network paths to deliberately delay or block message transmission. By intercepting proposals from the Leader and votes from Replica nodes, the attacker triggers timeout mechanisms, causing consensus rounds to fail.

In the ideal world, consensus behavior is modeled through the interaction of $\mathcal{F}_{\mathrm{HotStuff}}$ with $\mathcal{S}$. Before sending any consensus message (proposal, QC, vote), $\mathcal{F}_{\mathrm{HotStuff}}$ issues ⟨Sleep, phase$_p$, sid⟩ to $\mathcal{S}$ and receives ⟨Wake, sid, $\delta, \sigma\rangle$, where δ is the execution time and σ the adversarial delay. If $\delta + \sigma > \Delta^{\mathrm{vnum}}_{j}$ or $\sigma > \sigma$, an exponential backoff is triggered: $\Delta^{\mathrm{vnum}}_{j}$ is multiplied and a view change is enforced via $\mathcal{F}_{\mathrm{sync}}$; otherwise, consensus proceeds with $\mathcal{G}_{\mathrm{time}}$. This abstracts adversarial delay injections while bounding their power by $\sigma \leq \sigma$ and corruption of at most f nodes, ensuring indistinguishability of ideal and real executions.

In the Prepare phase, the Leader collects and verifies $2f + 1$ valid NEW-VIEW messages via $\mathcal{F}_{\mathrm{sig}}$ and $\mathcal{F}_{\mathrm{auth}}$, then broadcasts ⟨Prepare, CurProposal, highQC⟩ through $\mathcal{F}_{\mathrm{bc}}$. Replicas validate proposals with SafeNode$(m.\mathrm{node}, m.\mathrm{justify})$ and respond with VoteMsg⟨Prepare, m.node, $\perp\rangle$. A timeout from $\mathcal{G}$time aborts the phase and triggers recovery.

IV. UC Implementation

A. Formal Description of Real World

1) Modular Subroutines of Protocol π: To simplify formal representation and enable modular modeling, we decompose the HotStuff protocol into a collection of subroutines (or

subprotocols). These subroutines encapsulate specific operations such as message generation, message forwarding across different protocol phases, and the management of local consensus states. With this abstraction, the protocol π can be viewed as consisting of a main protocol augmented by a set of subprotocols.

This modularization not only streamlines the design but also facilitates composable security analysis under the UC framework. In particular, it provides a structured foundation for analyzing termination properties of the protocol in the presence of network delay attacks.

For example, we designed the $\mathcal{F}_{\text{propose}}$ function: This function collects NEW-VIEW messages from replica nodes using a threshold counter. When the number of valid NEW-VIEW messages reaches $2f + 1$, it selects the quorum proof (QC) with the highest height as the $highQC$ for the new proposal. It then binds the client command and leader identifier to the $highQC$ to generate a proposal, which is ultimately broadcasted using the broadcast ideal function $\mathcal{F}_{\text{bc}}$. Similarly, the $\mathcal{F}_{\text{QC}}$ function aggregates voting messages from each stage. After collecting $2f+1$ valid votes, it combines the fragmented signatures into a complete QC, which is then broadcasted using the broadcast ideal function $\mathcal{F}_{\text{bc}}$ to advance consensus.

Propose Subroutine $\mathcal{F}_{\text{propose}}$

Initialization: $proposal \leftarrow \perp$.

- Upon receiving $n - f$ NEW-VIEW message: $m_1, m_2, \ldots, m_{n-f}$ message, select the QC with the highest height as highQC:
  - $Determine\ highQC = max(QC_i \mid \forall m_i, QC_i.viewNumber)$
  - Write a client command to the leaf node of highQC to propose a new proposal B: $create\ proposal\ B = Leaf(highQC.node,\ command)$
  - Encapsulate proposal B and highQC in MSG and send it to the adversary $\mathcal{A}$: Get $\langle Prepare, sid, Curproposal, highQC\rangle$ from the adversary $\mathcal{A}$ and send it to the Leader

Fig. 10: Propose Subroutine $\mathcal{F}_{\text{propose}}$

2) the protocol π_{hotstuff}: In protocol π_{hotstuff}, the consensus process is organized into four chained phases—Prepare, PreCommit, Commit, and Decide, each building upon the quorum certificate (QC) generated in the preceding step. Specifically, once a leader is authenticated, it invokes $\mathcal{F}_{\text{propose}}$ to assemble a new proposal by binding the client command and its own identifier to the highest known QC, and then disseminates it to all replicas via $\mathcal{F}_{\text{bc}}$. Upon receiving the proposal, if correct, they cast votes that are collected and aggregated by $\mathcal{F}_{QC}$. When $2f + 1$ valid votes are gathered, $\mathcal{F}_{QC}$ produces a new QC, which not only justifies the leader's proposal but also advances the protocol into the next phase. This iterative mechanism ensures that once a CommitQC is formed, the protocol reaches the Decide phase, finalizes the decision, and outputs the client command.

QC Subroutine $\mathcal{F}_{\text{QC}}$

Initialization: $QC \leftarrow \perp$, Upon any $\langle \mathsf{timeOver}\rangle$ received from $\mathcal{G}_{\mathsf{time}}$, send NEW-VIEW message to $\mathcal{F}_{nextview}$.:

- Upon receiving 2f+1 valid voting messages $VOTEMSG(type, m.justiy.node, \perp)$:
  - First check whether m matches its own state: $MATCHINGMSG(m, type, curView)$
  - Collect the replicas' votes and combine the partial signatures:
    - $qc.type \leftarrow m.type \mid m \in V$
    - $qc.viewNumber \leftarrow m.viewNumber \mid m \in V$
    - $qc.node \leftarrow m.node \mid m \in V$
    - $qc.sig \leftarrow tcombine(qc.type, qc.viewNumber, qc.node, m.partialSig | m \in V)$
  - Encapsulate the QC in the MSG and send it to the adversary $\mathcal{A}$:
    - Get $\langle phase, \perp,\ phase.QC\rangle$ from the adversary$\mathcal{A}$, send it to $\mathcal{F}_{\text{bc}}$
- Upon a message $(\text{start}, \text{sid}, \text{curView})$ is received from the leader:
  - Send all messages in the message queue corresponding to curView to the leader.

Fig. 11: QC Subroutine $\mathcal{F}_{\text{QC}}$

Nextview Subroutine $\mathcal{F}_{\text{nextview}}$

Initialization: Set all counters to zero:

- Upon a NEW-VIEW request $\langle NEW-VIEW, \perp, PrepareQC\rangle$ is received from any replica r_i:
  - Store it in the message queue corresponding to the PrepareQC view number plus one.
- Upon a message $(\text{start}, \text{sid}, \text{curView})$ is received from the leader:
  - Send all messages in the message queue corresponding to curView to the leader.

Fig. 12: Nextview Subroutine $\mathcal{F}_{\text{nextview}}$

To handle view switching caused by node timeout, we designed a view switching function, $\mathcal{F}_{\text{nextview}}$. The specific design logic is that the view switching function $\mathcal{F}_{\text{nextview}}$ controls multiple message queues. Each message queue can store more than 3f+1 NEW-VIEW messages. When a node times out, it sends a NEW-VIEW message to $\mathcal{F}_{\text{nextview}}$.

$\mathcal{F}_{\text{nextview}}$ stores NEW-VIEW messages for the same view in the same queue. When a new leader requests a NEW-VIEW message, it sends the message in the message queue with the corresponding view number to the leader.

Vote Subroutine $\mathcal{F}_{\text{vote}}$

Initialization: Upon any ⟨timeOver⟩ received from $\mathcal{G}_{\text{time}}$, send NEW-VIEW message to $\mathcal{F}_{nextview}$.

- Upon receiving a $MSG(PREPARE, CurProposal, highQC)$ message from Leader:
  - First check whether message m matches its own state:
    - $MATCHINGMSG(m, PREPARE, curView)$
  - Check whether the leaf node is the successor of the local lockedQC corresponding node and whether the QC is higher than the view of the local lockedQC corresponding node:
    - Broadcast If $m.node$ extends from $m.justify.node \cap SAFENODE(m.node, m.justify)$.
- Upon receiving a $MSG(phase, \perp, phase.QC)$ message from Leader:
  - First check whether QC matches its own state: :
    - $MATCHINGQC(m.justify, phase, curView)\ from\ Leader(curView)$.
- If they match, the voting information node m and its own partial signature are encapsulated in VOTEMSG and sent to the adversary $\mathcal{A}$:
  - Get VOTEMSG $\langle phase_j, m.justify.node, \perp\rangle$and send it to Leader

Fig. 13: vote Subroutine $\mathcal{F}_{\text{vote}}$

To address replica node voting, we designed a voting ideal function, $\mathcal{F}_{vote}$, to implement state-driven verification logic. When a replica receives a leader proposal or QC, it first verifies the message's consistency with the current consensus stage (Prepare/PreCommit/Commit). It then performs security checks (such as whether the proposal conflicts with the locally locked QC). If these checks pass, a partially signed VOTEMSG is generated. This function dynamically binds message content, node state, and cryptographic credentials to ensure voting complies with both protocol stage constraints and the security commitments of chain extensions. Ultimately, legitimate votes are forwarded to the leader to advance the consensus process.

Meanwhile, to guard against delays or faulty leaders, $\mathcal{G}_{\text{time}}$ continuously tracks execution progress. If a timeout occurs, replicas trigger $\mathcal{F}_{\text{nextview}}$ to generate NEW-VIEW messages, which are again collected using a threshold counter; once $2f + 1$ valid NEW-VIEW messages are received, the system initiates a view change, elects a new leader, and resumes the four-phase pipeline. In this way, the protocol combines safety—achieved through the chained structure of QCs that prevent conflicting commits—with liveness—guaranteed under partial synchrony through timely view changes. The specific process is shown in Fig. 14.

The Protocol π_{hotstuff}

$\mathcal{Z}$	Leader	Replica	$\mathcal{A}$
→	1: Send ⟨timeStart, i, phase_p, δ⟩ to $\mathcal{G}_{\text{time}}$		
	2: Send $n - f$ NEW-VIEW messages to F_{propose}, and wait for the response ⟨Prepare, CurProposal, highQC⟩.		
	3: Send ⟨Input, sid, curView, cmd⟩ to $\mathcal{A}$.	⟶	
	4: Receive ⟨Input, sid, curView, cmd_A⟩ from $\mathcal{A}$.	⟵	
←	5: If receive ⟨timeOver, sid, phase_j, δ⟩ from G_{time}, send ⟨NEWVIEW, ⊥, PrepareQC⟩ to F_{nextview}.		
	6: Otherwise, broadcast ⟨Prepare, CurProposal, highQC⟩ to the replicas.	⟶	
		7: Send ⟨timeStart, i, phase_j, δ⟩ to $\mathcal{G}_{\text{time}}$	
←	Output ⟨Failure, sid, j⟩ to $\mathcal{Z}$	8: If receive ⟨timeOver, sid, phase_j, δ⟩ from G_{time}, send ⟨NEWVIEW, ⊥, PrepareQC⟩ to F_{nextview}.	
		9: Call $F_{vote}\langle Prepare, m.node, \perp\rangle$.	→ ←
←	10: If receive ⟨timeOver, sid, phase_j, δ⟩ from G_{time}, send ⟨NEWVIEW, ⊥, PrepareQC⟩ to F_{nextview}.		

Continued on next page

$\mathcal{Z}$	Leader	Replica	$\mathcal{A}$
	11: Call $\mathcal{F}_{QC}$ $\langle \text{PreCommit}, \perp, \text{PrepareQC} \rangle$	$\longrightarrow$	
$\leftarrow$	Output $\langle \mathsf{Failure}, \mathsf{sid}, j \rangle$ to $\mathcal{Z}$	12: If receive $\langle \text{timeOver}, \text{sid}, \text{phase}, \delta \rangle$ from G_{time}, send $\langle \text{NEWVIEW}, \perp, \text{PrepareQC} \rangle$ to F_{nextview}.	
		13: Call F_{vote} $\langle PreCommit, m.node, \perp \rangle$.	$\rightarrow$ $\leftarrow$
$\leftarrow$	14: If receive $\langle \text{timeOver}, \text{sid}, \text{phase}_j, \delta \rangle$ from G_{time}, send $\langle \text{NEWVIEW}, \perp, \text{PrepareQC} \rangle$ to F_{nextview}.		
	15: Call $\mathcal{F}_{QC}$ $\langle \text{Commit}, \perp, \text{PreCommitQC} \rangle$	$\longrightarrow$	
$\leftarrow$	Output $\langle \mathsf{Failure}, \mathsf{sid}, j \rangle$ to $\mathcal{Z}$	16: If receive $\langle \text{timeOver}, \text{sid}, \text{phase}, \delta \rangle$ from G_{time}, send $\langle \text{NEWVIEW}, \perp, \text{PrepareQC} \rangle$ to F_{nextview}.	
		17: Call F_{vote} $\langle Commit, m.node, \perp \rangle$.	$\rightarrow$ $\leftarrow$
$\leftarrow$	18: If receive $\langle \text{timeOver}, \text{sid}, \text{phase}_j, \delta \rangle$ from G_{time}, send $\langle \text{NEWVIEW}, \perp, \text{PrepareQC} \rangle$ to F_{nextview}.		
	19: Call $\mathcal{F}_{QC}$ $\langle \text{Decide}, \perp, \text{CommitQC} \rangle$	$\longrightarrow$	
$\leftarrow$	Output $\langle \mathsf{Failure}, \mathsf{sid}, j \rangle$ to $\mathcal{Z}$	20: If receive $\langle \text{timeOver sid}, \text{phase}, \delta \rangle$ from G_{time}, send $\langle \text{NEWVIEW}, \perp, \text{PrepareQC} \rangle$ to F_{nextview}.	

Continued on next page

$\mathcal{Z}$	Leader	Replica	$\mathcal{A}$
$\leftarrow$	Output $\langle \mathsf{Success}, \mathsf{sid}, id(v) \rangle$ to $\mathcal{Z}$	21: execute client's commad	
		22: send $\langle \text{NEWVIEW}, \perp, \text{PrepareQC} \rangle$ to F_{nextview}.	

Fig. 14: The Protocol π_{hotstuff}

B. UC Realization

Theorem 1 (π_{hotstuff} GUC Realizes $\mathcal{F}_{\mathsf{HotStuff}}$)**.** *The real world protocol π_{hotstuff} GUC realizes the ideal functionality $\mathcal{F}_{\mathsf{HotStuff}}$ under adversarial network delay attacks.*

Proof. We prove this theorem using game hopping. We start with the real protocol model and gradually modify it toward the final ideal function, ensuring that each modification is indistinguishable from the perspective of the environment $\mathcal{Z}$. We introduce a simulator $\mathcal{S}$ to intervene in the consensus communication network, introduce the protocol participants, the Leader and multiple Replica nodes, and construct a new ideal function $\mathcal{F}_{\text{HotStuff}}$ to connect these participants. Now let's modify the real protocol model step by step:

$\mathcal{F}_{\text{HotStuff}}$ introduces several state variables to track the replica's current view (currentView_i), phase ($\text{phase}i$), timeout counter (Δvnum^i), certificate verification status (cert_j), and a threshold counter for node vote counting ($\text{nums}_i\,(\text{NEW} - \text{VIEW})$). This modification only adds internal state records and does not change the message processing logic or external communication behavior, making it indistinguishable from $\mathcal{Z}$.

The adversary's delayed attack is mapped by $\mathcal{S}$ into the judgment that $\sigma^{L_i}_{\text{prepare}} > \Sigma$ in $\mathcal{F}\text{HotStuff}$, triggering a view increment. When $\mathcal{F}_{\text{time}}$ sends $\langle\text{timeOver}\rangle$, the view switching logic of $\mathcal{F}_{\text{HotStuff}}$ is triggered (NEW-VIEW broadcast and doubling Δ^i_{vnum}). The timeout mechanism is determined by thresholds of Δ^i_{vnum} and Σ in both the protocol and the ideal function, and $\mathcal{S}$ ensures that the timeout behavior of the two is synchronized, so $\mathcal{Z}$ cannot detect the difference.

After the view switch is triggered, $\mathcal{F}_{\text{sync}}\langle\text{RequestRound}\rangle$ is used to ensure that all timed-out nodes enter the next view at the same time. $\mathcal{F}_{\text{sync}}$ ensures that the node view switching operation is consistent with the real world, so $\mathcal{Z}$ cannot distinguish. When π_{hotstuff}'s $\mathcal{F}_{\text{QC}}$ collects $2f + 1$ votes, the built-in threshold counter ($\text{nums}_i\,(\text{phase})$) in $\mathcal{F}_{\text{HotStuff}}$ reaches the threshold, generating the corresponding Prepare/PreCommit/Commit messages.The QC generation and verification logic is identical in the protocol and ideal function, and the adversary's attack is fully simulated by $\mathcal{S}$, making it indistinguishable to $\mathcal{Z}$. From state synchronization, message processing, timeout mechanisms, QC generation, to final

submission, the simulator $\mathcal{S}$ ensures behavioral equivalence between the real protocol and the ideal function at every step. Any environment $\mathcal{Z}$ cannot distinguish between the two by observing communication, attack effects, or protocol outputs. Therefore, the protocol π_{hotstuff} implements the ideal function $\mathcal{F}_{\text{HotStuff}}$. □

V. UC Termination Proof

In this section, we show that $\mathcal{F}_{\mathsf{HotStuff}}$ is guaranteed to terminate. Together with Theorem 1, we can derive the termination of π_{hotstuff}.

Theorem 2 (HotStuff Protocol UC Termination)**.** *In the HotStuff protocol, assuming the adversary controls at most f nodes (including the possible leader) and the message delay is bounded ($\delta_{\text{adv}} < 2^{f+1}\Delta$), then there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{Z}$:*

$$T^* = \mathsf{GST} + O(f^2\Delta) \tag{1}$$

Then, for any block height h and any honest process $p \in \mathcal{H}$, when the system time $t \geq T^*$, the protocol π_{hotstuff} must terminate at height h and output a unique decision value v_h: $\mathsf{decide}_h(p) = v_h$ *and v_h is consistent across all $p \in \mathcal{H}$.*

The environment $\mathcal{Z}$ cannot distinguish between the execution and output of the real protocol π_{hotstuff} and the ideal function $\mathcal{F}_{\text{HotStuff}}$.

In the following, we first provide some characteristics of HotStuff, and the capability of adversary as well. Then, we give the proof of Theorem 2.

A. HotStuff Protocol Characteristics and Adversary Model

HotStuff mitigates network latency through an exponential backoff strategy, where the timeout parameter doubles each round, $\Delta^{(r)} = 2 \cdot \Delta^{(r-1)}$, allowing adaptive wait times and reducing unnecessary view switches. Consensus progresses through three phases—Prepare, PreCommit, Commit—each requiring at least $2f + 1$ valid votes.

The Simulator $\mathcal{S}$ controls up to f nodes and can issue malicious proposals or votes via protocol interfaces, attempting to disrupt phase consistency. Message delivery can be delayed by at most δ_{adv}, but cannot be blocked indefinitely, ensuring eventual delivery. Moreover, $\mathcal{S}$ cannot forge signatures of more than f honest nodes, preserving the Byzantine bound $n \geq 3f + 1$ and guaranteeing termination under the UC framework.

B. Scenario by Scenario Termination Proof

We now prove Theorem 2, i.e, the termination of π_{HotStuff}, under arbitrary adversarial delay σ_{adv}, by considering both the real world execution and its ideal world simulation $\mathcal{F}_{\text{HotStuff}}$.

1) Case 1(bounded delay $\sigma_{adv} \leq \Delta$): If adversarial delay is within the timeout window, each of the four phases (Prepare, PreCommit, Commit, Decide) completes within Δ. For each phase p, the leader (possibly corrupted) broadcasts the message m_p at time $t_0 + \sigma_p^L$, honest replicas respond within $[t_0, t_0 + \Delta]$, and corrupted replicas delay at most $\sigma_p^R \leq \Delta$. Since $2f + 1$ valid votes are always collected, a QC is generated in each phase.In the ideal world, simulator $\mathcal{S}$ schedules identical delays via $\langle \text{Wake}, sid, p, \delta, \sigma_p^R \rangle$ and triggers GENERATE QC(p) once $2f + 1$ votes are observed.Thus, both worlds produce the same message sequence $\langle \text{Prepare}, CurProposal, highQC \rangle$, $\langle \text{PreCommit}, \perp, PrepareQC \rangle$, $\langle \text{Commit}, \perp, PreCommitQC \rangle$, $\langle \text{Decide}, \perp, CommitQC \rangle$, all before $t_0 + \Delta$. The environment $\mathcal{Z}$ observes no difference.

2) Case 2 (excess delay $\sigma_{adv} > \Delta$): If delay exceeds the timeout, replicas detect expiration at Δ^i_{vnum}, broadcast $\langle \text{NEW-VIEW}, curView + 1 \rangle$, and double the timeout parameter: $\Delta^i_{vnum} \leftarrow 2\Delta^i_{vnum}$. After at most f consecutive faulty views, the $(f + 1)$-th view is led by an honest leader (due to round robin leader election).In the ideal world, $\mathcal{S}$ simulates timeouts through $\mathcal{F}_{time}$, increments $curView$, updates Δ^i_{vnum}, and resynchronizes all replicas via $\mathcal{F}_{sync}$. Both worlds, therefore, exhibit the same pattern of view changes, doubling of timeouts, and QC chains.

3) Case 3 (first honest view $curView = f + 1$): At the first honest leader, at least $n - f$ NEW-VIEW messages are collected, each containing a latest $highQC$. By the SafeNode rule, the leader selects a block B justified by the maximal $highQC$, broadcasts $\langle \text{Prepare}, B, highQC \rangle$, and all honest replicas accept since $highQC.view \geq LockedQC.view$. The protocol then deterministically progresses through PreCommit, Commit, and Decide, generating final decision within bounded time $T_f = 3 \cdot 2^{f+1}\Delta$.In both worlds, the QC chain is unique and irreversible, guaranteeing consensus termination.

The worst case total termination time is

$$t^* = \sum_{k=0}^{f} 2^k\Delta + 3 \cdot 2^{f+1}\Delta = O(2^f\Delta).$$

This covers adversarial delay $\sigma_{\text{adv}} \leq 2^{f+1}\Delta$

For each message m, its delivery delay in the real world δ_{real} and in the ideal world δ_{ideal} satisfy:

$$|\delta_{\text{real}} - \delta_{\text{ideal}}| = 0 \quad (\sigma_{\text{adv}} \leq 2^{f+1}\Delta)$$

since excess delays trigger identical timeout driven view changes in both worlds. Furthermore, invalid proposals and votes are filtered by $\mathcal{F}_{bc}$ and $\mathcal{F}_{vote}$ in the same way. Hence, the environment $\mathcal{Z}$ observes identical proposal chains, QC sequences, and decision outputs in both worlds.Conclusion.HotStuff satisfies UC termination: under any adversarial delay $\sigma_{\text{adv}} \leq 2^{f+1}\Delta$, all honest replicas decide on the same value within $t^* = O(2^f\Delta)$, and $\mathcal{Z}$ cannot distinguish π_{hotstuff} from $\mathcal{F}_{\text{HotStuff}}$.

VI. Conclusion and Future Work

This paper provides a UC based termination analysis of HotStuff by formalizing its ideal functionality $\mathcal{F}_{\mathrm{Hotstuff}}$ and real protocol $\pi\mathrm{hotstuff}$, proving indistinguishability under network delay attacks. Modular decomposition of authentication, timing, and voting, together with the exponential timeout $\Delta^{(r)} = 2^{r-1}\Delta$, bounds adversarial view switches to $O(2^{f-1}\Delta)$ and ensures consensus within $2^f\Delta$. The quantitative relation between Δ and fault tolerance f guides latency estimation and node sizing. Limitations include partial synchrony assumptions and static adversaries; future work targets dynamic networks and more complex delays.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," vol. 4, no. 3, 1982, doi: 10.1145/357172.357176.

[2] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988, doi: 10.1145/42282.42283.

[3] X. Xu, I. Weber, and M. Staples, "Architecture for blockchain applications," 2019.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, p. 173–186.

[6] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM symposium on principles of distributed computing*, 2019, pp. 347–356, doi: 10.1145/3293611.3331591.

[7] Y. Lu, Z. Lu, and Q. Tang, "Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2159–2173, doi: 10.1145/3548606.3559346.

[8] G. Kola, T. Kosar, and M. Livny, "Faults in large distributed systems and what we can do about them," in *European Conference on Parallel Processing*. Springer, 2005, pp. 442–453, doi: 10.1007/11549468_51.

[9] M. Treaster, "A survey of fault-tolerance and fault-recovery techniques in parallel systems," *arXiv preprint cs/0501002*, 2005.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985, doi: 10.1145/3149.214121.

[11] 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," *The Libra Assn., Tech. Rep*, vol. 7, 2019.

[12] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145, doi: 10.1145/34024571.

[13] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42, doi: 10.1145/2976749.2978399.

[14] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[15] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: A scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 568–580, doi: 10.1109/DSN.2019.00063.

[16] V. Shoup, "Practical threshold signatures," in *International conference on the theory and applications of cryptographic techniques*. Springer, 2000, pp. 207–220.

[17] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002, doi: 10.1145/571637.571640.

[18] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2011, doi: 10.1109/TC.2011.221.

[19] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 59–74, 2005, doi: 10.1145/1095809.1095817.

[20] L. Jehl, "Formal verification of hotstuff," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2021, pp. 197–204.

[21] L. Qiu, Y. Kim, J.-Y. Shin, J. Kim, W. Honoré, and Z. Shao, "Lido: Linearizable byzantine distributed objects with refinement-based liveness proofs," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1140–1164, 2024, doi: 10.1145/3656423.

[22] S. Duan and H. Zhang, "Foundations of dynamic bft," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1317–1334.

[23] R. Saltini and D. Hyland-Wood, "Correctness analysis of ibft," *arXiv preprint arXiv:1901.07160*, 2019.

[24] F. Gai, C. Grajales, J. Niu, M. M. Jalalzai, and C. Feng, "A secure consensus protocol for sidechains," *arXiv preprint arXiv:1906.06490*, 2019.

[25] R. Cohen, P. Forghani, J. Garay, R. Patel, and V. Zikas, "Concurrent asynchronous byzantine agreement in expected-constant rounds, revisited," in *Theory of Cryptography Conference*. Springer, 2023, pp. 422–451.

[26] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography Conference*. Springer, 2007, pp. 61–85.

[27] A. Kiayias, M. Kohlweiss, and A. Sarencheh, "Peredi: Privacy-enhanced, regulated and distributed central bank digital currencies," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1739–1752.

[28] D. Achenbach, J. Müller-Quade, and J. Rill, "Synchronous universally composable computer networks," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 95–111.

[29] R. Canetti, "Universally composable signature, certification, and authentication," in *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* IEEE, 2004, pp. 219–233.