Consensus in the Known Participation Model with Byzantine Failures and Sleepy Replicas

Chenxu Wang*, Sisi Duan†‡✉, Minghui Xu*✉, Feng Li*, and Xiuzhen Cheng*
*Shandong University, †Tsinghua University
‡State Key Laboratory of Cryptography and Digital Economy Security
cxwang16117@gmail.com, duansisi@tsinghua.edu.cn, {mhxu,fli,xzcheng}@sdu.edu.cn
✉ Corresponding authors

***Abstract*—We study consensus in the known participation model with both Byzantine failures and sleepy replicas, where honest replicas may unpredictably fall asleep, and replicas know the minimum number of awake honest replicas. Our main contribution is providing a fine-grained t reatment o f consensus in such a mixed failure model. First, we present a synchronous atomic broadcast protocol with $5\Delta + 2\delta$ expected latency and $2\Delta + 2\delta$ best-case latency, where Δ is the bound on network delay and δ is the actual network delay. Second, in the partially synchronous network (the value of Δ is unknown), we show that one can make a conventional Byzantine fault-tolerant (BFT) protocol tolerate sleepy replicas but has to make the stable storage assumption (where replicas need to store intermediate consensus parameters in stable storage). Finally, in the partially synchronous network but not assuming stable storage, we show several bounds on the relationship between the total number of replicas n, the maximum number of Byzantine replicas f, and the maximum number of simultaneous sleeping replicas s. Using these bounds, we transform HotStuff (PODC'19) into a protocol that tolerates sleepy replicas without sacrificing the performance.**

I. Introduction

Byzantine fault-tolerant state machine replication (BFT) is a fundamental tool in fault-tolerant distributed computing, allowing a group of replicas to reach an agreement in the presence of arbitrary failures [1]–[7]. Conventional BFT protocols assume that replicas know each other's identities and non-faulty replicas are expected to always stay online. With the rise of Bitcoin [8], BFT (and blockchains) have been characterized by new features. The *sleepy model* of consensus [9] is an example. In sleepy consensus, besides Byzantine replicas, honest replicas may unpredictably go offline (and become *asleep*) and later come back online (and become *awake*). So far, all the sleepy consensus protocols focus on the unknown participation model [9]–[13], where replicas are not aware of the minimum number of awake honest replicas h_a. Sleepy consensus protocols can only be built in a synchronous network [9]–[13], where there exists a known upper bound for message processing and propagation. Pass and Shi [9] have shown an impossibility result for partially synchronous and asynchronous networks. Such protocols are also called *dynamically available* protocols [14]–[17].

Another notable study is consensus with a mixed failure model, i.e., the system consists of both Byzantine failures and other types of failures. Such a study has a history of more than a decade, and many prior works consider both Byzantine and crash failures [18]–[21]. In fact, sleepy consensus can also be viewed as one in the mixed failure model. For example, Momose and Ren (MR) [10] present a model where sleepy replicas are embedded with a *recovery* module. Sleepy model with recovery matches the *crash-recovery* model in the distributed computing literature [22], where replicas keep infinitely often crashing and recovering. MR provides a sleepy consensus protocol under this model and describes the protocol as one that "*tolerates any number of crash-recovery faults plus minority Byzantine faults* [10, Sec. 3.1]."

Since sleepy consensus focuses on the unknown participation model, we make a clear distinction in this work. We call replicas that may fall asleep and become awake *sleepy replicas*. We use sleepy consensus to denote protocols that tolerate both Byzantine failures and an unknown number of sleepy replicas. In this work, we are interested in protocols in the known participation model, where the minimum number of awake honest replicas h_a is known. We call our protocols ones that tolerate both Byzantine failures and sleepy replicas in the known participation model and show that these protocols are of independent interest.

We provide a comparison of both sleepy consensus and consensus in the known participation model in Table I. As summarized in the table, consensus in the known participation model has not been well studied.

Motivation of the known participation model. Consensus with both Byzantine failures and sleepy replicas in the known participation model is useful and has been informally studied in existing systems. One example is Ethereum [23]. In the current production system, Ethereum assumes the partially synchronous model and does not handle sleepy replicas by design. It expects honest replicas (called *validators*) to remain awake and use the *incentive mechanism* to penalize inactive replicas. However, it was shown that an attack could cause *all* honest replicas to be penalized even if honest replicas strictly

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230448
www.ndss-symposium.org

TABLE I: **Comparison between sleepy consensus and consensus in the known participation model with sleepy replicas.**

Network	Participation model	Protocol	Minimum n to tolerate s asleep replicas	Maximum number of asleep honest replicas	Expected latency	Stable storage?
Synchronous	Unknown (i.e., sleepy consensus)	MR [10] (without the recovery protocol)	$n = 2f + s + 1$	$n - 2f - 1$	32Δ	✓
		MR [10] (with the recovery protocol)	$n = 2f + s + 1$	$n - 2f - 1$	35Δ	✗
		MMR [12]	$n = 2f + s + 1^{\ddagger}$	$n - 2f - 1^{\ddagger}$	14Δ	✗
	Known	**Koala-1 (Sec. III)**	$n = 2f + s + 1$	$n - 2f - 1$	$5\Delta + 2\delta$	✗
Partially synchronous	Unknown	Impossible. [9]	–	–	–	–
	Known	a sub-protocol of* [17] **with stable storage (Sec. IV)**	$n = 3f + 1^{\dagger}$	$n - f^{\dagger}$	7Δ (based on HotStuff)	**Required**
		Koala-2 (Sec. V)	$n = 3f + s + 1^{\dagger}$ or $n = 3f + 2s + 1$	$n - 3f - 1^{\dagger}$ or $\lfloor \frac{n-3f-1}{2} \rfloor$	7Δ (based on HotStuff)	✗

n is the number of replicas, f is the maximum number of Byzantine replicas, and s is the maximum number of asleep honest replicas. Δ is the upper bound for message processing and transmission latency and δ is the actual network latency. ‡The bound considers the worst case where all f Byzantine replicas remain awake. *The Ebb-and-Flow protocol has a sub-protocol Π_{bft} that tolerates Byzantine failures and sleepy replicas. Π_{bft} does not specify whether stable storage is required. We show that the result can only be achieved by assuming stable storage. Namely, with stable storage, all $n - f$ honest replicas can fall asleep before the global awake time (GAT). † Liveness of the protocol is guaranteed under the GAT assumption, where all honest replicas remain awake after GAT.

follow the specification of the protocol [24]. Accordingly, consensus in the known participation model that handles sleepy replicas by design can systematically address this issue.

Another example is that industrial systems such as Diem-BFT [25] and Tendermint [26] are already taking actions to handle sleepy replicas but in an informal way. For instance, the DiemBFT codebase [27] clearly mentions that some parameters are stored in the stable storage to "ensure liveness even if all replicas crash and later recover" (persistent_liveness_storage.rs:24-28[1]). However, storing *all* consensus parameters might be too expensive [22], [28], [29]. In the literature of BFT research, no analysis is given to identify what should be stored in stable storage.

Therefore, an interesting open problem is:

Can we provide a more fine-grained treatment of consensus in the known participation model with both Byzantine failures and sleepy replicas?

Our technical approaches. In this paper, we provide a fine-grained treatment of consensus in the known participation model (where the minimum number of awake honest replicas h_a is known) in both synchronous and partially synchronous networks, with and without stable storage. As summarized in Table I, we provide the following results.

▷ **Koala-1: fast synchronous atomic broadcast with Byzantine failures and sleepy replicas**. While we can directly use a sleepy consensus protocol in the unknown participation model, the latency of the protocol can be very high. We show that in the known participation model, the latency can be made closer to conventional consensus protocols. Our protocol requires $n \geq 2f + s + 1$ (i.e., $h_a = f + 1$), where n is the number of replicas, f is the maximum number of Byzantine replicas, and s is the maximum number of asleep honest replicas. An interesting fact is that existing sleepy consensus protocols also assume $n \geq 2f + s + 1$ [9]–[13] but f and s are unknown.

We present Koala-1 that achieves an expected latency of $5\Delta + 2\delta$ and a best-case latency of $2\Delta + 2\delta$, where Δ is the upper bound for message processing and propagation latency and δ is the actual network latency. In contrast, the best result of sleepy consensus has a latency of 14Δ (i.e., MMR [12]), more than twice the latency of Koala-1. Our result is closer to the $2\Delta+\delta$ latency achieved by conventional synchronous BFT protocols [30] (not in the sleepy model). The major challenge we address is that the conventional *Byzantine quorum* does not work in the model where some replicas may fall asleep. Important building blocks of Koala-1 include a new *double confirmation mechanism* and a new *validated triple-graded proposal election (VT-GPE)* primitive, which might be of independent interest.

▷ **Partially synchronous consensus with stable storage and the impossibility result.** To date, the only known result of consensus with both Byzantine replicas and sleepy replicas in the partially synchronous network is a Π_{bft} sub-protocol of Ebb-and-Flow [17] and its follow-up work [31]. It was briefly mentioned that one can use conventional partially synchronous BFT protocols [5], [7], [32] to support sleepy replicas by assuming global awake time (GAT), where after GAT, every honest replica becomes awake. However, concrete constructions are not provided.

In this work, we show that by assuming the conventional $n = 3f+1$ bound, a partially synchronous consensus protocol that tolerates Byzantine and sleepy replicas cannot be achieved without making the stable storage assumption. We provide

[1]DiemBFT: https://github.com/diem/diem/blob/3b774462384ea37dd46f1c67b525f03937c45a0a/consensus/src/persistent_liveness_storage.rs

formal proof of the impossibility result, and use HotStuff [7] as a case study to show our results. While storing all the intermediate parameters in stable storage is an option, it is usually very expensive to do so as frequent disk I/O is involved [22], [28], [29]. For instance, it was shown that the throughput of BFT-SMaRt [33] is only 23% of its storage-free counterpart [29]. We then show that we only need to store two parameters (the *view number* and $lockedQC$) to make HotStuff tolerate sleepy replicas (denoted as HotStuff-mSS). We consider our result a complement to BFT in production systems and the Π_{bft} sub-protocol of Ebb-and-Flow [17].

▷ **Koala-2: partially synchronous protocol without stable storage.** In the partially synchronous model without assuming stable storage, we transform HotStuff into Koala-2, a protocol that tolerates Byzantine and sleepy replicas and still retains the 7δ latency of HotStuff. we require that $s = \lfloor \frac{n-3f-1}{2} \rfloor$, i.e., $n \geq 3f+2s+1$ and $h_a = \lceil \frac{n+f+1}{2} \rceil$. (As defined, the values of h_a and s can be calculated from each other. Given s, h_a can be determined as $n-f-s$, and vice versa). Additionally, by assuming the existence of GAT, the value of s can be further improved to $n-3f-1$ (i.e., $n \geq 3f+s+1$ and $h_a = 2f+1$). We show that these bounds are tight.

Our contributions. Our contributions are summarized below:

- (Sec. III) In the synchronous model, we present Koala-1, a protocol that tolerates both Byzantine and sleepy replicas in the known participation model. Koala-1 achieves an expected latency of $5\Delta+2\delta$, almost half of the latency of the state-of-the-art sleepy consensus protocol (in the unknown participation model).
- (Sec. IV) In the partially synchronous model, we show that a BFT protocol with the $n = 3f+1$ bound can not tolerate sleepy replicas without assuming stable storage. We further use HotStuff as an example and show that by assuming GAT, we only need to store two parameters in stable storage to tolerate sleepy replicas.
- (Sec. V) In the partially synchronous model, we propose Koala-2, a HotStuff-variant that tolerates sleepy replicas without assuming stable storage while still achieving the same latency as HotStuff. The bound is $s = \lfloor \frac{n-3f-1}{2} \rfloor$ without the GAT assumption and $s = n-3f-1$ with the GAT assumption.
- (Sec. VI) We evaluate the performance of HotStuff-mSS and Koala-2 and show that both protocols are efficient, achieving performance very close to HotStuff (where no intermediate parameters are stored in stable storage). Additionally, we also provide a demo, showing that HotStuff suffers from the *double spending attack* with no stable storage. In contrast, none of our protocols suffers from the attack.

II. System Model and Building Blocks

Byzantine fault tolerance (BFT). In a BFT protocol, clients *submit* transactions (requests) and replicas *deliver* them. The client obtains a final response to the submitted transaction from the responses. Within a BFT system of n replicas, a maximum of f replicas may fail arbitrarily under the control of an adversary. These faulty replicas are also known as Byzantine failures and non-Byzantine replicas are called *honest* replicas. The correctness of a BFT protocol is specified as follows:

- **Safety**: If an honest replica *delivers* a transaction tx before *delivering* tx', no honest replica *delivers* the transaction tx' without first *delivering* tx.
- **Liveness**: If a transaction tx is *submitted* to all honest replicas, all awake honest replicas eventually *deliver* tx.

An equivalent primitive atomic broadcast (ABC) is often used interchangeably with BFT. Informally, the main difference is that ABC does not involve the role of the clients.

- **Safety**: If an honest replica *a-delivers* a message m before it *a-delivers* m', then no honest replica *a-delivers* the message m' without first *a-delivering* m.
- **Liveness**: If an honest replica *a-broadcasts* a message m, then all awake honest replicas eventually *a-deliver* m.

While the BFT and atomic broadcast abstractions do not expose the *order* to API, an implicit order is given in most protocols, e.g., sequence number [5], [34], height [7], [35]. Using this implicit order, many partially synchronous protocols achieve a weaker safety property as follows [5], [7], [35].

- **Consistency**: If an honest replica *delivers* a transaction tx and another honest replica *delivers* a transaction tx', both with the same order, $tx = tx'$.

Our Koala-1 protocol follows the conventional atomic broadcast model. Our Koala-2 protocol achieves the consistency property, following that of HotStuff.

Network models and communication channels. We consider both synchronous and partially synchronous networks. In the synchronous model, there exists an upper bound Δ for message processing and transmission latency and there exists a completely synchronous clock. In the partially synchronous model [36], there still exists an upper bound but the value of Δ is unknown. An alternative notion of the partially synchronous model is that there exists an unknown global stabilization time (GST) such that after GST, messages sent between two honest replicas arrive within a fixed delay. Note that an asynchronous network does not assume such an upper bound.

We assume authenticated channels for message transmission. We use the symbol $*$ to denote any value. We use δ to denote the actual network latency.

Sleepy replicas. A sleepy replica can be either *awake* or *asleep* [9]. An awake replica actively participates in the execution, while an asleep replica does not execute any code of the protocol or send/receive any message. In our system, each honest replica can become asleep, whose status can change at any time under the control of an adversary, without any advance notice. In practice, this implies that replicas are allowed to leave and rejoin the protocol's execution at will without notifying other replicas. Sleepy replicas align with the crash-recovery model, where replicas can keep crashing and recovering repeatedly [22]. It is highlighted that an honest replica might encounter "amnesia" after crashing, leading to

the loss of its internal state stored in its volatile storage. In our work, Byzantine replicas can also be sleepy.

Our work considers the known participation model, where all replicas have foreknowledge of the minimum number of awake honest replicas h_a. Meanwhile, we use s to denote the maximum number of asleep replicas at any point of the protocol execution. As defined, h_a and s can be calculated from each other. Given s, h_a can be determined as $n-f-s$, and vice versa. In our synchronous protocol, h_a is $f+1$. In our partially synchronous protocol, h_a is $\lceil \frac{n+f+1}{2} \rceil$. If we consider the global awake time (GAT) assumption, where after GAT every sleeping replica will be awake, our partially synchronous protocol can be achieved with $h_a = 2f+1$.

Cryptographic assumptions. We use digital signatures, making the public-key infrastructure (PKI) assumption. We use $\langle \mu \rangle_i$ to denote a message μ signed by replica p_i. We assume a cryptographic collision-resistant hash function denoted as $\mathrm{H}(\cdot)$. We assume a verifiable random function (VRF) in one of our protocols. A replica p_i evaluates $(\rho_i, \pi_i) \leftarrow \mathrm{VRF}_i(\mu)$ on any input μ and obtains a pseudorandom value ρ_i and a proof π_i. Using π_i and the public key of replica p_i, anyone can verify whether ρ_i is a correct evaluation of VRF_i on μ.

Blocks. We use *block* B to denote a batch of transactions. Blocks are ordered in a chain where the previous block of B is called its *parent block*. The first block in the chain is called the genesis block B_0. A block B extends block B' if B' is an ancestor of B in the chain. Two blocks B and B' conflict with each other if neither of them extends the other.

Byzantine quorums and quorum certificates. A *byzantine quorum* (or *quorum* in short) denotes certain number of replicas. Matching *votes* from a quorum is necessary for honest replicas to reach an agreement. A set of signatures signed by a quorum of replicas is called a *quorum certificate* (QC or *certificate*). In conventional BFT with $n \geq 3f+1$ replicas, a Byzantine quorum consists of $\lceil \frac{n+f+1}{2} \rceil$ replicas. By slightly abusing notation, we use the $view()$ function to denote the view number of a QC or a block. For example, if qc is a QC for block B, $view(qc)$=$view(B)$.

Graded proposal election (GPE). In GPE [12], each replica *gpe-proposes* a block and *gpe-decides* either (B, g) or $\perp$, where B is a block and $g \in \{0, 1\}$ is the grade. GPE achieves the following properties:

- **Consistency.** If an honest replica *gpe-decides* $(B, *)$ and another honest replica *gpe-decides* $(B', *)$, $B = B'$.
- **Graded delivery.** If an honest replica *gpe-decides* $(B, 1)$, all honest replicas *gpe-decide* $(B, *)$.
- $1/2$**-validity.** With a probability of at least 1/2, all honest replicas *gpe-decide* $(B, 1)$, where B has been *gpe-proposed* by an honest replica.

III. Koala-1: Fast Synchronous Consensus with Byzantine and Sleepy Replicas

We introduce a synchronous atomic broadcast protocol called Koala-1 that tolerates both Byzantine and sleepy replicas. We consider a system with $n \geq 2f+s+1$ replicas and $h_a = f+1$. Without loss of generality, we assume stable storage and message delivery, i.e., once a replica becomes awake at time t, it will immediately receive all messages sent from any honest replica before time $t-\Delta$. Later in Appendix B, we provide a practical recovery protocol to remove both assumptions.

TABLE II: **Comparison of synchronous BFT protocols.**

Protocol	Failure model	Participation Model	Expected latency	Best-case latency
Sync-HotStuff [37]	Byzantine	Known	$2\Delta+\delta$	$2\Delta+\delta$
MR [10]	Byzantine & sleepy	Unknown	32Δ	16Δ
MMR [12]		Unknown	14Δ	4Δ
Koala-1		Known	$5\Delta+2\delta$	$2\Delta+2\delta$

Δ is the upper bound on message processing and transmission latency and δ is the actual network latency.

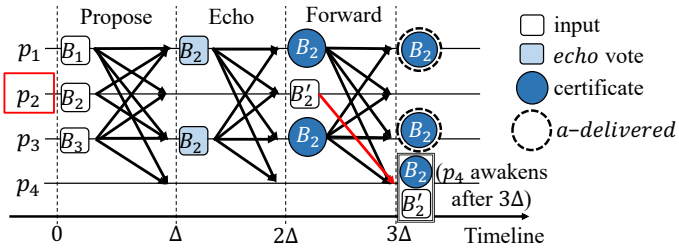
We build a practical protocol with latency close to conventional synchronous BFT protocols (e.g., Sync HotStuff [37] has $2\Delta+\delta$ latency). In particular, Koala-1 has a fast path that achieves $2\Delta+2\delta$ latency, which occurs when all awake replicas are honest. The result is much lower than sleepy consensus, as summarized in Table II.

A. Overview of Koala-1

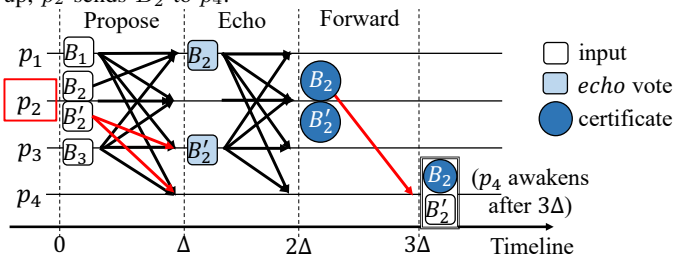
In the classic static participation model, one can obtain a synchronous atomic broadcast (ABC) protocol assuming $n = 2f+1$ and a quorum size of $f+1$. This is because an quorum certificate (QC) with $f+1$ votes is *transferrable* and can be verified by any replica. Most prior synchronous Byzantine agreement and atomic broadcast protocols [30], [37], [38] all follow a *commit-lock* paradigm, i.e., once a replica commits a block B, all honest replicas will be locked on B. This paradigm ensures that honest replicas will only vote for blocks extending B. In this way, the safety of the protocol is guaranteed. Such a paradigm can be realized via the GPE protocol as reviewed in Sec. II.

In the mixed failure model, a tempting solution is to change the size of the Byzantine quorum and *transform* the protocol to one that tolerates sleepy replicas. Unfortunately, even under the known participation model, building a secure protocol is not trivial. This is mainly because we can no longer use the conventional quorum size as (possibly more than the majority of) honest replicas may become asleep and lose their state. In fact, even if a certificate with h_a matching votes is transferrable (e.g., under the help of a powerful equivocation detection mechanism), there might still be safety and liveness issues. We show two scenarios in Figure 1. In both scenarios, p_2 is a Byzantine leader but it equivocates in different ways. From the perspective of an honest replica p_4, the two scenarios are indistinguishable. Thus, we need to design the protocol using additional techniques.

In Koala-1, our main contribution is h_a*-enabled quorum*, i.e., using h_a as the quorum size and make the certificate with h_a matching votes transferrable. This is achieved via



(a) Scenario 1: p_2 sends B_2 to p_1 and p_3. p_1 and p_3 echo B_2, collect a certificate, and forward the certificate. As p_1 and p_3 detect no equivocation before $t = 3\Delta$, they *a-deliver* B_2. When p_4 wakes up, p_2 sends B'_2 to p_4.



(b) Scenario 2: p_2 sends B_2 to p_1 and B'_2 to p_3 and p_4. p_1 echoes B_2 and p_3 echoes B'_2. As p_1 and p_3 observe the equivocation, none of p_1 or p_3 *a-deliver* any block.

Fig. 1: Two situations of synchronous ABC protocols with sleepy replicas that are indistinguishable for p_4. In both scenarios, p_2 is Byzantine, p_4 receives B'_2 from p_2 and a valid certificate for B_2. In scenario 1, p_1 and p_3 *a-deliver* B_2. In contrast, in scenario 2, none of p_1 or p_3 *a-deliver* any block.

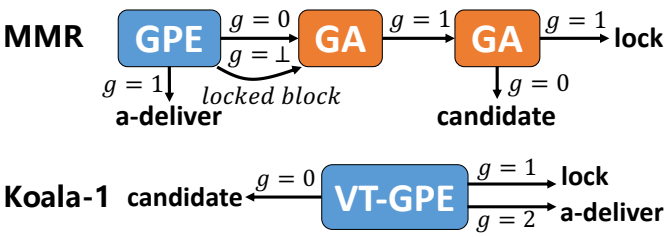


Fig. 2: MMR [12] vs. Koala-1. GA denotes graded agreement.

a carefully designed *double confirmation mechanism* and an equivocation mechanism. The double confirmation mechanism uses two certificates as a *proof* for a block B to make the proof verifiable. Meanwhile, we also extend the GPE notion to validated triple-graded proposal election (VT-GPE). As illustrated in Figure 2, compared to prior work, our protocol is much simplified.

B. Validated Triple-Graded Proposal Election

Validated Triple-graded Proposal Election (VT-GPE). We define *validated triple-graded proposal election* (VT-GPE) as follows. Each honest replica *tgpe-proposes* a block (together with a valid proof) and *tgpe-decides* either (B, g, σ) (where $B \neq \perp$) or $\perp$. Here, g is a grade where $g \in \{0, 1, 2\}$. We also need an *external validity* property for VT-GPE to be validated. In particular, we define a global predicate that is determined by the particular application and known to all parties. In this work, we define σ as the proof for the validity of block B. Let the predicate be Q and we say B is validated by σ if $Q(B, \sigma)$ holds. Each honest replica only *tgpe-decides* one block in a VT-GPE instance, but it may *tgpe-decides* the same block multiple times with different grades. A validated VT-GPE protocol achieves the following properties:

- **External validity.** If an honest replica *tgpe-decides* $(B, *, *)$ such that $B \neq \perp$, $Q(B, \sigma)$ holds for at least an honest replica.
- **Consistency.** If an honest replica *tgpe-decides* $(B, *, *)$ and another honest replica *tgpe-decides* $(B', *, *)$, $B = B'$.
- **Graded delivery.** If an honest replica *tgpe-decides* $(B, g, *)$ such that $g \in \{1, 2\}$, any honest replica *tgpe-decides* $(B, g-1, *)$.
- **Validity.** With a probability of $\alpha > 1/2$, all honest replicas *tgpe-decide* $(B, 2, *)$ where block B is *tgpe-proposed* by an honest replica.

The VT-GPE protocol. We use VT-GPE$_v$ to denote a VT-GPE instance. VT-GPE$_v$ consists of two phases: a VRF-based leader election phase and a graded consensus phase. The leader election phase selects a leader and honest replicas may select different leaders. The graded consensus phase allows replicas to converge on the result of the leader election.

As shown in Algorithm 1, the protocol begins with a VRF-based leader election. Each replica p_i broadcasts a $\langle \text{INPUT}, B_i, \sigma_i, \rho_i, \pi_i \rangle_i$ message, where B_i is the block p_i *tgpe-proposes*, σ_i is the proof for B_i, ρ_i is a VRF evaluation on the current view number, and π_i is a proof of the VRF evaluation. As defined above, every replica only considers B_i valid if $Q(B_i, \sigma_i)$ holds. For now we do not care about the instantiation of σ_i and later we will define it. The VRF evaluations are used for leader election. In particular, according to the VRF evaluations each replica receives, the producer of the highest VRF is considered the leader, and the corresponding $\langle \text{INPUT} \rangle$ message is defined as the *winning input*. If equivocating $\langle \text{INPUT} \rangle$ messages are received from the leader, the winning input is set as $\perp$. Additionally, the block B associated with the winning input is called the *winning block*.

As each replica may receive different sets of $\langle \text{INPUT} \rangle$ messages and the winning inputs might be different, we define the winning input for each replica p_i. In particular, $\langle \text{INPUT}, B, \sigma, \rho, \pi \rangle_j$ from p_j is a winning input for p_i if the following conditions are met:

(1) $Q(B, \sigma)$ holds;
(2) π is a valid proof of ρ on the current view number;
(3) ρ is the highest among all the VRF evaluations in the $\langle \text{INPUT} \rangle$ messages;
(4) p_i has not received another valid $\langle \text{INPUT}, B', \sigma', \rho, \pi \rangle_j$ such that $B' \neq B$.

After the leader election, from time $t = \Delta$ to $t = 4\Delta$, the graded consensus phase is executed as follows.

- At $t = \Delta$, if replica p_i is awake, it broadcasts an $\langle \text{ECHO} \rangle$ message for the winning block.

Algorithm 1 Validated Triple-graded Proposal Election of view v - VT-GPE$_v$.

Replica p_i executes the following algorithm at every time $t \geq 0$ after starting VT-GPE$_v$ in view v, and *tgpe-proposes* (B_i, σ_i) such that a global predicate $Q(B_i, \sigma_i)$ holds.
p_i maintains these parameters for each received block B:
    $E(B) \leftarrow$ all received $\langle \text{ECHO}, B \rangle_*$ messages
    $R(B) \leftarrow$ all received $\langle \text{READY}, B \rangle_*$ messages
    $L(B) \leftarrow$ all received $\langle \text{LOCK}, B \rangle_*$ messages
    $W_1(B) \leftarrow$ all received $\langle \text{WINNER1}, \langle \text{INPUT}, B \rangle_* \rangle_*$ messages
    $W_2(B) \leftarrow$ all received $\langle \text{WINNER2}, \langle \text{INPUT}, B \rangle_* \rangle_*$ messages
if $t = 0$ **then**
    $(\rho_i, \pi_i) \leftarrow \text{VRF}_i(v)$
    broadcast $\langle \text{INPUT}, B_i, \sigma_i, \rho_i, \pi_i \rangle_i$
if $t = \Delta$ **then**
    if there exists a winning input $\langle \text{INPUT}, B_j, \sigma_j, \rho_j, \pi_j \rangle_j$ **then**
        forward the winning input (if not yet)
        if $Q(B_j, \sigma_j)$ holds **then**
            broadcast $\langle \text{ECHO}, B_j \rangle_i$
    else
        forward the equivocating INPUT messages by any replica
if $t = 2\Delta$ **then**
    update local winning input from received INPUT messages
    if $\langle \text{INPUT} \rangle_j \neq \perp$ **then** // Let $\langle \text{INPUT} \rangle_j$ be the winning input
        broadcast $\left\langle \text{WINNER1}, \langle \text{INPUT} \rangle_j \right\rangle_i$
        if $|E(B_j)| \geq f + 1$ **then**
            broadcast $E(B_j)$ and $\langle \text{READY}, B_j \rangle_i$
    else
        forward the equivocating INPUT messages by any replica
if $t = 3\Delta$ **then**
    update local winning input from received INPUT messages
    if $\langle \text{INPUT} \rangle_j \neq \perp$ **then**
        broadcast $\left\langle \text{WINNER2}, \langle \text{INPUT} \rangle_j \right\rangle_i$
        if $|R(B_j)| \geq f + 1$ **then**
            broadcast $R(B_j)$ and $\langle \text{LOCK}, B_j \rangle_i$
    else
        forward the equivocating INPUT messages by any replica
if $t \geq 4\Delta$ **then**
    update local winning input from received INPUT messages
    if $\langle \text{INPUT} \rangle_j \neq \perp$ **and** $|L(B_j)| \geq f + 1$ **then**
        tgpe-decide $(B_j, 2, L(B_j))$
    if $|R(B)| \geq f+1$ **and** $|W_2(B)| \geq f+1$ for a block B **then**
        tgpe-decide $(B, 1, (R(B), W_2(B)))$
    if $|E(B)| \geq f+1$ **and** $|W_1(B)| \geq f+1$ for a block B **then**
        tgpe-decide $(B, 0, (E(B), W_1(B)))$
    if no block is *tgpe-decided* **then**
        tgpe-decide $\perp$

- At $t = 2\Delta$, p_i broadcasts a $\langle \text{WINNER1} \rangle$ message containing its winning input. If p_i receives at least $f + 1$ matching $\langle \text{ECHO} \rangle$ messages for its winning block, the replica forwards these $\langle \text{ECHO} \rangle$ messages and broadcasts a $\langle \text{READY} \rangle$ message.
- At $t = 3\Delta$, p_i broadcasts a $\langle \text{WINNER2} \rangle$ message containing its winning input. Similar to the previous round, if p_i receives at least $f + 1$ $\langle \text{READY} \rangle$ messages for its winning block, the replica forwards these $\langle \text{READY} \rangle$ messages and broadcasts a $\langle \text{LOCK} \rangle$ message.
- When $t \geq 4\Delta$, there are four conditions. First, if p_i receives $f + 1$ matching $\langle \text{LOCK} \rangle$ messages for its winning block B_j, it *tgpe-decides* B_j with grade 2 and uses $f + 1$ $\langle \text{LOCK} \rangle$ messages as the proof for B_j. Second, if p_i receives $f + 1$ $\langle \text{READY} \rangle$ and $f+1$ $\langle \text{WINNER2} \rangle$ messages for any block B, it *tgpe-decides* B with grade 1. Here, both $f+1$ $\langle \text{READY} \rangle$ and $f+1$ $\langle \text{WINNER2} \rangle$ messages are used as proofs for B. Finally, if p_i receives $f + 1$ $\langle \text{ECHO} \rangle$ and $f + 1$ $\langle \text{WINNER1} \rangle$ messages for any block B, it *tgpe-decides* B with grade 0. Here, the $\langle \text{ECHO} \rangle$ and $\langle \text{WINNER1} \rangle$ messages are used as proofs for B. Otherwise, p_i *tgpe-decides* a special symbol $\perp$.

C. Atomic Broadcast (ABC)

Our ABC protocol follows the view-by-view construction of many classic BFT protocols [5], [7], [35] and also prior sleepy consensus protocols. In each view, each honest replica *a-broadcasts* a block and *a-delivers* at most one block.

The protocol starts from view 1 and the pseudocode for view v is shown in Algorithm 2. In each view v, there is one VT-GPE instance denoted as VT-GPE$_v$. In each VT-GPE$_v$, each replica p_i *tgpe-proposes* block B that extends its candidate, where B is the block p_i *a-broadcasts*. Here, our idea is to use the grade $g \in \{2, 1, 0\}$ of VT-GPE to mimic the *commit-lock-prepare* relation in conventional BFT. To maintain the status, every replica maintains several local parameters, including the candidate and lock, which are initially set as the genesis block B_0. If a block B is *tgpe-decided* with grade 0 (resp. 1), the candidate (resp. lock) is set as B.

We define the global predicate Q for VT-GPE as follows. Given the value (B, qc) *tgpe-proposed* by any replica p_j, $Q(B, qc)$ holds at p_i if and only if:

- $view(B)$ equals the current view number of p_i, qc is a valid $prepareQC$ for B, and B's parent block is the block of qc;
- the view number of qc is at least the same as p_i's lock.

In our protocol, $prepareQC$ is the *proof* each replica p_i holds after it *tgpe-decides* a block B with grade 0. According to our VT-GPE instantiation, the proof consists of two certificates, i.e., $f + 1$ $\langle \text{ECHO} \rangle$ messages and $f + 1$ $\langle \text{WINNER1} \rangle$ messages for B. The certificates are crucial for B to be validated and we call them the double confirmation mechanism for B. Meanwhile, ensuring the view number of qc is at least the same as p_i's locked block further prevents forks from happening and is crucial for both safety and liveness.

Every replica p_i waits for the output of VT-GPE$_v$ and there are three possible outputs.

(1) If p_i *tgpe-decides* $(B, 0, (E(B), W_1(B)))$, p_i sets its candidate as B and $prepareQC$ as $(E(B), W_1(B))$.
(2) If p_i *tgpe-decides* $(B, 1, (R(B), W_2(B)))$, it sets its lock as B and $lockedQC$ as $(R(B), W_2(B))$. A valid $lockedQC$ for block B consists of $f+1$ $\langle \text{READY} \rangle$ and $f+1$ $\langle \text{WINNER2} \rangle$ messages for B. The lock parameter is useful for defining the predicate Q and the $lockedQC$ parameter is only useful in the recovery protocol (to be described in Appendix B).
(3) If p_i *tgpe-decides* $(B, 2, L(B))$, it *a-delivers* B and all the ancestors of B. It also sets its $commitQC$ as $L(B)$, which is only useful in the recovery protocol.

Algorithm 2 The Koala-1 atomic broadcast protocol for p_i.

Initialize the following parameters
  $v \leftarrow 1$; candidate $\leftarrow B_0$; lock $\leftarrow B_0$; $prepareQC \leftarrow \perp$; $lockedQC \leftarrow \perp$; $commitQC \leftarrow \perp$. // $lockedQC$ and $commitQC$ are used in the recovery protocol
Let Q be the following predicate for VT-GPE:
  Given (B, qc) *tgpe-proposed* by p_j,
  $Q(B, qc) \equiv (view(B) = v)$ **and** (qc is a valid $prepareQC$)
  and $(B.parent = qc.block)$ **and** $view(qc) \geq view(\mathsf{lock})$
In each view v, replica p_i executes the following algorithm at every time $0 \leq t \leq 4\Delta$ w.r.t. view v, then enters view $v+1$.
if $t = 0$ **then**
  $B \leftarrow \langle \mathrm{val}s, \mathrm{H}(\mathsf{candidate}), v \rangle_i$
  tgpe-propose $(B, prepareQC)$ in VT-GPE$_v$ with predicate Q
// The following events may be triggered after view v
upon p_i *tgpe-decides* $(B, 0, (E(B), W_1(B)))$ in VT-GPE$_v$ **do**
  if $view(B) > view(\mathsf{candidate})$ **then**
    candidate $\leftarrow B$, $prepareQC \leftarrow (E(B), W_1(B))$
upon p_i *tgpe-decides* $(B, 1, (R(B), W_2(B)))$ in VT-GPE$_v$ **do**
  if $view(B) > view(\mathsf{lock})$ **then**
    lock $\leftarrow B$, $lockedQC \leftarrow (R(B), W_2(B))$
upon p_i *tgpe-decides* $(B, 2, L(B))$ in VT-GPE$_v$ **do**
  if B has not been *a-delivered* **then**
    a-deliver B and all ancestors of B, $commitQC \leftarrow L(B)$

Pipelining mode. Our protocol enjoys the benefit of pipelining, where replicas can enter the next view $v+1$ at $t = 3\Delta$ of the current view v. While a new instance VT-GPE$_{v+1}$ is started, the current instance VT-GPE$_v$ still runs until each replica *tgpe-decides*. To see why replicas can enter the next view at $t = 3\Delta$, consider that an honest replica is locked on a block B in VT-GPE$_v$. All replicas awake at $t = 3\Delta$ must receive the $prepareQC$ (including $f+1$ $\langle \text{ECHO} \rangle$ and $f+1$ $\langle \text{WINNER1} \rangle$ messages) for B. Any honest replica that proposes new blocks must extend B in newer views. Besides, as lock can be updated at $t = 4\Delta$ of view v, replicas can use their updated lock to verify the new blocks at $t = \Delta$ of view $v+1$.

Fast path. Our protocol has a fast path that *a-delivers* a block in $2\Delta+2\delta$ time. We achieve this by slightly modifying our VT-GPE primitive into a weaker version called wT-GPE. wT-GPE no longer achieves the consistency property and has a *weak consistency* property instead, defined as follows.

- *Weak consistency.* If an honest replica *tgpe-decides* $(B, g, *)$ with grade $g \geq 1$ and another honest replica *tgpe-decides* $(B', *, *)$, $B = B'$.

The weak consistency property of VT-GPE achieves consistency only if an honest replica *tgpe-decides* a block with a grade of at least 1. Via this change, we do not need the $\langle \text{WINNER1} \rangle$ and $\langle \text{WINNER2} \rangle$ messages in our wT-GPE construction. Accordingly, each replica *tgpe-decides* a block B with grade 0 when it receives valid $E(B)$ at $t \geq 3\Delta$. If there are multiple such blocks, choose the one corresponding to the highest VRF. Meanwhile, each replica *tgpe-decides* a block B with grade 1 or 2 after it receives valid $R(B)$ or $L(B)$ at time $t > 2\Delta$.

Algorithm 3 Validated Triple-graded Proposal Election with *Weak Consistency* for view v - wT-GPE$_v$.

Replica p_i executes the following algorithm at every time $t \geq 0$ after starting wT-GPE$_v$ in view v, and *tgpe-proposes* (B_i, σ_i) such that a global predicate $Q(B_i, \sigma_i)$ holds.
p_i maintains these parameters for each received block B:
  $E(B) \leftarrow$ all received $\langle \text{ECHO}, B \rangle_*$ messages
  $R(B) \leftarrow$ all received $\langle \text{READY}, B \rangle_*$ messages
  $L(B) \leftarrow$ all received $\langle \text{LOCK}, B \rangle_*$ messages
if $t = 0$ **then**
  $(\rho_i, \pi_i) \leftarrow \mathrm{VRF}_i(v)$
  broadcast $\langle \text{INPUT}, B_i, \sigma_i, \rho_i, \pi_i \rangle_i$
if $t = \Delta$ **then**
  if there exists a winning input $\langle \text{INPUT}, B_j, \sigma_j, \rho_j, \pi_j \rangle_j$ **then**
    forward the winning input (if not yet)
    if $Q(B_j, \sigma_j)$ holds **then**
      broadcast $\langle \text{ECHO}, B_j \rangle_i$
  else
    forward the equivocating INPUT messages by any replica
if $t = 2\Delta$ **then**
  update local winning input from received INPUT messages
  if $\langle \text{INPUT} \rangle_j \neq \perp$ **then** // Let $\langle \text{INPUT} \rangle_j$ be the winning input
    forward $\langle \text{INPUT} \rangle_j$ (if not yet)
    if $|E(B_j)| \geq f+1$ **then**
      broadcast $E(B_j)$ and $\langle \text{READY}, B_j \rangle_i$
if $2\Delta < t \leq 3\Delta$ **then**
  if $|R(B)| \geq f+1$ for any block B **then**
    broadcast $R(B)$ and $\langle \text{LOCK}, B \rangle_i$ (if not yet)
if $t > 2\Delta$ **then**
  if $|L(B)| \geq f+1$ for any block B **then**
    tgpe-decide $(B, 2, L(B))$
  if $|R(B)| \geq f+1$ for any block B **then**
    tgpe-decide $(B, 1, R(B))$
if $t \geq 3\Delta$ **then**
  for each $\langle \text{INPUT}, B_j, \sigma_j, \rho_j, \pi_j \rangle_j$ **do** // from inputs with higher ρ_j
    if $|E(B_j)| \geq f+1$ **then**
      tgpe-decide $(B_j, 0, E(B_j))$
      break
  if no block is *tgpe-decided* **then**
    tgpe-decide $\perp$

Although we do not need the $\langle \text{WINNER1} \rangle$ and $\langle \text{WINNER2} \rangle$ messages, our wT-GPE protocol still employs the double confirmation mechanism to make $prepareQC$ consistent with $lockedQC$ in each view. This is achieved by additionally modifying the predicate Q. In particular, upon receiving a valid $prepareQC$ qc with $view(qc) = view(\mathsf{lock})$, each replica additionally checks whether the block of the $prepareQC$ is the same as its lock. In this way, only the $prepareQC$ that matches the $lockedQC$ will be verified by each honest replica.

We present the pseudocode of the wT-GPE in Algorithm 3 and the pseudocode of our pipelined Koala-1 protocol (with the fast path) in Algorithm 4.

D. Analysis

Why h_a-enabled quorum? The double confirmation mechanism we use ensures that a certificate with h_a matching messages is transferrable. In our VT-GPE construction, we

Algorithm 4 The pipelined ABC protocol. Code for p_i.

Initialize the following parameters
  $v \leftarrow 1$; candidate $\leftarrow B_0$; lock $\leftarrow B_0$; $prepareQC \leftarrow \perp$; $lockedQC \leftarrow \perp$; $commitQC \leftarrow \perp$. // $lockedQC$ and $commitQC$ are used in the recovery protocol
Let Q be the following predicate for wT-GPE:
  Given (B, qc) *tgpe-proposed* by p_j,
  $Q(B, qc) \equiv (view(B) = v)$ **and** (qc is a valid $prepareQC$)
  and $(B.parent = qc.block)$ **and**
  $(view(qc) > view(\mathsf{lock})$ **or** $qc.block = \mathsf{lock})$
In each view v, replica p_i executes the following algorithm at every time $0 \leq t \leq 3\Delta$ w.r.t. view v, and then enter view $v+1$.
if $t = 0$ **then**
  $B \leftarrow \langle \text{vals}, \text{H}(\mathsf{candidate}), v \rangle_i$
  tgpe-propose $(B, prepareQC)$ in $\mathsf{wT\text{-}GPE}_v$ with predicate Q
// The following events may be triggered after view v
upon p_i *tgpe-decides* $(B, 0, E(B))$ in $\mathsf{wT\text{-}GPE}_v$ **do**
  if $view(B) > view(\mathsf{candidate})$ **then**
    candidate $\leftarrow B$, $prepareQC \leftarrow E(B)$
upon p_i *tgpe-decides* $(B, 1, R(B))$ in $\mathsf{wT\text{-}GPE}_v$ **do**
  if $view(B) > view(\mathsf{lock})$ **then**
    lock $\leftarrow B$, $lockedQC \leftarrow R(B)$
upon p_i *tgpe-decides* $(B, 2, L(B))$ in $\mathsf{wT\text{-}GPE}_v$ **do**
  if B has not been *a-delivered* **then**
    a-deliver B and all ancestors of B, $commitQC \leftarrow L(B)$

use the double confirmation scheme for both grade 0 and grade 1. To *tgpe-decide* block B with grade 0, a replica needs to collect $f+1$ matching $\langle \text{ECHO} \rangle$ messages and $f+1$ matching $\langle \text{WINNER1} \rangle$ messages for B. Meanwhile, to *tgpe-decide* B with grade 1, a replica needs to collect $f+1$ matching $\langle \text{READY} \rangle$ messages and $f+1$ matching $\langle \text{WINNER2} \rangle$ messages for B.

In our protocol, we can distinguish the two scenarios for p_4 in Figure 1. Namely, we introduce one change on top of the toy construction: each replica additionally broadcasts a $\langle \text{WINNER1} \rangle$ message at $t = 2\Delta$ for the block from the leader. In scenario 1 (Figure 1a), p_1 and p_3 do not detect any equivocation, so they send $\langle \text{WINNER1} \rangle$ messages for block B_2 at $t = 2\Delta$. When p_4 wakes up after 3Δ, it receives the $\langle \text{WINNER1} \rangle$ messages due to the message delivery assumption. Therefore, p_4 can now *gpe-decide* B_2 with grade 0 according to the double confirmation mechanism. Now consider scenario 2 (Figure 1b), p_1 and p_3 detect the equivocation after receiving both B_2 and B_2', so none of them sends a $\langle \text{WINNER1} \rangle$ message. As p_4 has not received $\langle \text{WINNER1} \rangle$, it does not *gpe-decide* B_2. We show the proof of Koala-1 in Appendix A and the correctness of pipelined Koala-1 (with the fast path) in our full paper [39].

Latency and complexity. In the fast path, the latency of our ABC protocol is $2\Delta + 2\delta$. Specifically, replicas need to wait for Δ time in the first two communication rounds of each wT-GPE instance. In the last two rounds, each replica can proceed to the next round after collecting a sufficient number of messages. Meanwhile, the expected latency is $5\Delta + 2\delta$, as a block is expected to be *a-delivered* every two views and each replica enters the next view as early as 3Δ time has elapsed. Also, Koala-1 achieves $O(\kappa n^3 + Ln^2)$ communication, where κ is the security parameter and L is the size of a block.

IV. Partially Synchronous Protocol with Stable Storage

In this section, we focus on the partially synchronous model assuming the existence of stable storage. As mentioned in the introduction, a sub-protocol in Ebb-and-Flow briefly mentions that by assuming GAT, one can use a conventional BFT [5], [7], [32] to obtain a protocol that tolerates sleepy replicas in the partially synchronous model. Namely, conventional BFT protocols are safe in the presence of sleepy replicas and live after both GAT and GST.

We show that the above statement can be achieved *only* if stable storage is assumed and *intermediate* consensus parameters are stored in stable storage. To date, most BFT protocols known so far do not explicitly discuss what should be stored in stable storage as it is usually out of the scope of the consensus problem. We show that without explicitly storing the intermediate parameters, conventional BFT may not be safe and live in the presence of sleepy replicas while retaining the $n \geq 3f+1$ assumption, even assuming both GST and GAT. Intuitively, this is because if an honest replica does not persist its intermediate status during the protocol, its *status* might not be resumed after it sleeps and later becomes awake. Even if the replica synchronizes with all honest replicas after it becomes awake, the protocol may still not be correct.

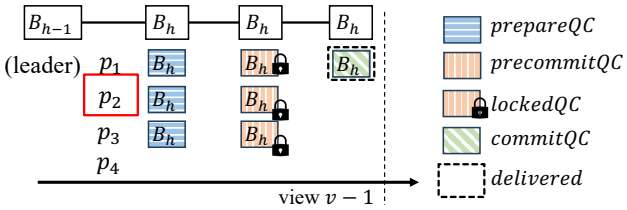
In particular, we prove the following impossibility result and we show the proof in Appendix C.

Theorem 1. *In the partially synchronous model, any BFT protocol cannot handle sleepy replicas under the $n \geq 3f+1$ assumption without the stable storage, where f is the number of byzantine replicas and n is the total number of replicas.*

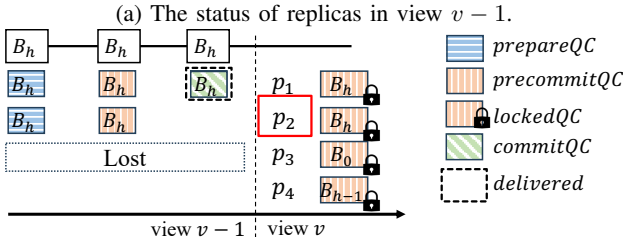
In this section, we use HotStuff as a case study and show an attack on safety without assuming stable storage. Due to the space limitation, we provide an overview of HotStuff in our full paper [39]. We show that while one can simply make conventional BFT tolerate sleepy replicas by asking each replica to store *all* intermediate parameters in stable storage, we offer a much cheaper approach where only two parameters need to be stored. As stable storage is expensive [22], [28], [29], our approach can then complement BFT in production systems in terms of supporting sleepy replicas.

A. A Case Study of HotStuff

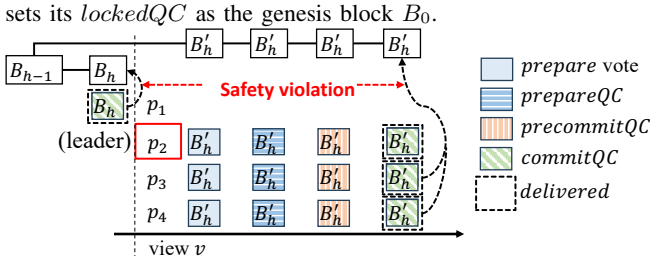
We present a case study of HotStuff in Figure 3 with four replicas among which p_2 is faulty. We show that if no stable storage is assumed, safety can be violated. In the period of asynchrony, we consider that an adversary (i.e., a network scheduler) *manipulates* the network, the same as the assumption made by asynchronous protocols [1], [2], [6]. Note that in a partially synchronous network, we can assume the existence of a network scheduler during the asynchronous period. However, the network becomes synchronous after GST. Additionally, the adversary *controls* the replicas that may



(a) The status of replicas in view $v-1$.



(b) p_3 goes to sleep near the end of view $v-1$ and becomes awake at the beginning of view v. It *loses* its *lockedQC* and sets its *lockedQC* as the genesis block B_0.



(c) The faulty leader p_2 creates a fork that extends B_{h-1} in view v and is able to collect a QC with votes from p_2, p_3, and p_4. Safety is violated as p_1 delivers B_h and p_3 and p_4 deliver B'_h.

Fig. 3: A case study of HotStuff. In particular, with no stable storage (no intermediate consensus parameters are stored in stable storage), HotStuff does not achieve safety in the presence of sleepy replicas.

become asleep. In this case, the asleep replicas are still honest but just cannot process any messages when they sleep. Under these assumptions, the attack proceeds as follows.

In view $v-1$, as shown in Figure 3a, p_1 is the leader and it proposes block B_h. After p_1 collects a *commitQC*, it delivers block B_h and replicas p_1, p_2, and p_3 become locked on B_h. Here, the network scheduler delays the messages received by p_4. Therefore, although p_4 is honest, it has not received any messages for B_h. After that, p_3 becomes asleep.

As shown in Figure 3b, replicas then enter view v and p_2 becomes the leader. p_3 becomes awake in view v. As p_3 does not have stable storage, it *loses* its *lockedQC*. As a result, *lockedQC* is set as the genesis block B_0. In view v, the leader p_2 is faulty and proposes a new block B'_h that extends B_{h-1} (the parent block of B_{h-1} is B_h). As B'_h is conflicting with B_h, p_1 considers the proposal B'_h invalid and will not vote for B'_h. However, p_2, p_3, and p_4 can vote for B'_h, as p_2 is faulty and the *lockedQC* of p_3 and p_4 is not conflicting with B'_h.

Finally, as illustrated in Figure 3c, replica p_1 delivers block B_h and replicas p_3 and p_4 deliver block B'_h where B'_h and B_h are conflicting, violating the safety property of the protocol.

Remark 1. We assume that the adversary manipulates the network and the replicas that go to sleep. In practice, even if the adversary does not manipulate the network and the replicas that go to sleep, the scenarios may still happen, e.g., during network asynchrony or server crash.

Remark 2. Although we present a concrete example using HotStuff, almost *all* partially synchronous BFT protocols use a variant of *commit-lock-prepare* paradigm [5], [32], [35]. Our attack is thus generic to almost all (if not all) partially synchronous BFT. Since we already provide a formal proof of the impossibility result, we omit the generalization of the attack in our paper.

B. HotStuff-mSS: A Fully-Fledged Protocol under the Stable Storage Assumption

Based on the discussion above, a natural solution to build a protocol under the $n \geq 3f+1$ assumption is storing *all* intermediate consensus parameters in stable storage. However, the system performance might also be degraded significantly. Therefore, an interesting research question to answer is:

Under the $n \geq 3f+1$ and stable storage assumption, can we transform a conventional BFT protocol to one that tolerates sleepy replicas and stores minimum intermediate consensus parameters in stable storage?

We use HotStuff as an example and show that the minimum requirement for stable storage is the view number and the *lockedQC*. We use HotStuff-mSS to denote this protocol. Namely, if the current view number is lost when an honest replica falls asleep, the replica can only *catch up* with other replicas to learn the latest view number after waking up. It is possible that the replica re-enters the same view before it fell asleep. In this case, the replica might vote for a conflicting block with the one it has voted for (before falling asleep). Thus, two conflicting qc could be generated in the same view, violating the safety property. To ensure safety within a view, the highest view v when a replica has cast a vote should be stored in stable storage.

Meanwhile, the attack described in Figure 3 shows that safety across views might be violated if a replica loses its *lockedQC*. We show that storing *lockedQC* in stable storage is sufficient to ensure safety across views. In particular, if a block B is delivered, a quorum of replicas becomes locked on B. To deliver a block B' conflicting with B, at least one honest replica of the quorum must have voted for B'. Since the honest replica already sets its *lockedQC* as B, it will never vote for a block conflicting with B. Consider the example mentioned in Figure 3. p_3 stores its *lockedQC* for block B_h in stable storage before it goes asleep. When p_3 becomes awake at the beginning of view v, it restores *lockedQC* for B_h from its stable storage. In view v, leader p_2 proposes a new block B'_h that extends B_{h-1} (B_h's parent block). As B'_h is conflicting with B_h, replica p_1 and p_3 do not consider the proposal B'_h valid and will not vote for B'_h. Thus, only p_2 and p_4 can vote for B'_h. *prepareQC*, *precommitQC* or *commitQC* cannot be formed for B'_h, so no honest replicas will deliver block B'_h.

V. KOALA-2: PARTIALLY SYNCHRONOUS PROTOCOL WITHOUT STABLE STORAGE

We study partially synchronous protocols that tolerate sleepy replicas without the stable storage assumption. We require $n \geq 3f + 2s + 1$ if GAT is not assumed, and $n \geq 3f + s + 1$ if GAT is assumed. We discuss below that the bounds are tight.

When GAT is not assumed, $n \geq 3f+2s+1$ and a Byzantine quorum of $\lceil \frac{n+f+1}{2} \rceil$ is sufficient. This is because every two Byzantine quorums have at least one overlapped honest and awake replica. Also, the number of awake replicas must be equal to or greater than the quorum size, i.e., $n - f - s \geq \lceil \frac{n+f+1}{2} \rceil$. Meanwhile, if GAT is assumed, the bound can be lowered to $n \geq 3f + s + 1$. This is mainly because under the GAT assumption, there exists some point after which all honest replicas are awake. Let β_1 be the quorum size and β_2 be the number of messages a recovering replica needs to receive. The upper bounds of β_1 and β_2 are: $\beta_1 \leq n - f$ and $\beta_2 \leq n - f - s$. To ensure safety, β_1 and β_2 should have at least $f + 1$ overlapped replicas. In this way, at least one honest replica is in common. Thus, $\beta_1 + \beta_2 - n \geq f + 1$, i.e., $n \geq 3f + s + 1$.

Still using HotStuff as an example, we transform the protocol into Koala-2, a BFT protocol that tolerates sleepy replicas. The main workflow remains almost the same as in HotStuff. We only need to adjust the quorum size of the main protocol and modify the view change protocol (i.e., leader election) to incorporate a $timeoutQC$ mechanism. Besides, we introduce a new recovery protocol for asleep replicas to catch up after they recover. An asleep replica first enters the *recovering* status and completes the recovery protocol before it becomes awake.

A. Overview of Koala-2

We now present the Koala-2 protocol without the GAT assumption. Given the bound on n, f, and s, we only need to change the quorum size of HotStuff from $n - f$ to $n - f - s$. The main technical challenge we solve is to ensure that any honest replica that recovers will vote for the correct block(s). This is not easy, as asleep replicas may become awake at any time. Without stable storage, we need to ensure all recovered replicas maintain the *latest* state so safety is not violated.

We provide a modified view change protocol and a recovery protocol to achieve the goal above. The technical building blocks include a $timeoutQC$ mechanism and an *atomic QC acquiring* mechanism. The $timeoutQC$ mechanism ensures that a recovering replica obtains at least the state in the view it fell asleep. Meanwhile, the atomic QC acquiring mechanism ensures that each recovering replica obtains the *highest* QC that is necessary to ensure the safety of the system.

B. The Modified View Change Protocol and the Recovery Protocol

In this section, we present the modified view change protocol (Algorithm 5) and our new recovery protocol (Algorithm 6).

Algorithm 5 Modified view change protocol (for replica p_i).

1: Let cv be the current view number.
2: **upon** the timer of cv expires **do**
3: broadcast $\langle \text{TIMEOUT}, cv \rangle_i$
4: **upon** receiving $f + 1$ $\langle \text{TIMEOUT}, cv \rangle_*$ **do**
5: stop the timer of cv and broadcast $\langle \text{TIMEOUT}, cv \rangle_i$
6: **upon** receiving $n - f - s$ $\langle \text{TIMEOUT}, v' \rangle_*$ such that $v' \geq cv$ **do**
7: $timeoutQC \leftarrow$ the set of $n - f - s$ $\langle \text{TIMEOUT}, v' \rangle_*$
8: broadcast $\langle \text{ADVANCE-VIEW}, v', timeoutQC \rangle_i$
9: send $\langle \text{NEW-VIEW}, v'+1, prepareQC \rangle_i$ to leader of view $v'+1$
10: $cv \leftarrow v' + 1$
11: **upon** receiving a $timeoutQC$ tc of a view $v' \geq cv$ **do**
12: $timeoutQC \leftarrow tc$
13: broadcast $\langle \text{ADVANCE-VIEW}, v', tc \rangle_i$
14: send $\langle \text{NEW-VIEW}, v'+1, prepareQC \rangle_i$ to leader of view $v'+1$
15: $cv \leftarrow v' + 1$

The modified view change protocol. The modified view change protocol is triggered when a timeout occurs during the normal case operation. When a replica p_i experiences a timeout in a view v, it stops the normal case operation and broadcasts a $\langle \text{TIMEOUT}, v \rangle_i$ message. A collection of $n - f - s$ matching $\langle \text{TIMEOUT} \rangle$ messages from different replicas forms a $timeoutQC$. After receiving a $timeoutQC$ of view v, p_i enters view $v+1$. To expedite the view change process, p_i broadcasts the $\langle \text{TIMEOUT}, v \rangle_i$ message once receiving $f + 1$ $\langle \text{TIMEOUT} \rangle$ messages of view v. When p_i receives the $timeoutQC$ of view v, it forwards the $timeoutQC$ to all replicas.

The recovery protocol. The protocol proceeds as follows:

- **Obtaining** $timeoutQC$**.** A recovering replica p_i first sends $\langle \text{RECOVERY-1} \rangle$ to all replicas. Upon receiving the $\langle \text{RECOVERY-1} \rangle$ message, any awake replica will respond to p_i the latest $timeoutQC$ (via a $\langle \text{ECHO-1} \rangle$ message). Once receiving $n - f - s$ $timeoutQC$, p_i selects the one with the highest view number v_h. Then p_i waits for a $timeoutQC$ of a view $v \geq v_h + 2$ before entering the next step.
- **Atomic QC acquiring mechanism.** After receiving a $timeoutQC$ tc for a view $v \geq v_h + 2$, p_i sets its local $timeoutQC$ as tc, and sends $\langle \text{RECOVERY-2}, tc \rangle_i$ to all replicas. Any awake replica that receives this message will first start the view change protocol and proceed to view $view(tc) + 1$ (if not yet). Then the replica sends to p_i a $\langle \text{ECHO-2}, cv, (prepareQC, lockedQC, commitQC) \rangle$ message, where cv is the current view number. Also, the replica sends to p_i all delivered blocks, the block B corresponding to $prepareQC$, and all ancestors of B. When p_i receives $n - f - s$ valid $\langle \text{ECHO-2} \rangle$ with view numbers higher than v_h+2, it sets its $lockedQC$ as the highest $lockedQC$ among the messages, sets its $prepareQC$ as the highest $prepareQC$, and sets its $commitQC$ as the highest $commitQC$. Then, p_i delivers the block corresponding to $commitQC$ and all its ancestors. p_i also sets the current view number as $view(timeoutQC) + 1$ and wakes up.

Correctness and complexity. We prove the correctness of Koala-2 in our full paper [39] and sketch the correctness here. Safety is guaranteed via the $timeoutQC$ mechanism and the

Algorithm 6 Recovery protocol for HotStuff (for replica p_i).

Let cv be the current view number, pqc be the $prepareQC$, lqc be the $lockedQC$, and cqc be the $commitQC$.
as a recovering replica
  broadcast a $\langle \text{RECOVERY-1} \rangle_i$ message
  wait for $n - f - s$ $\langle \text{ECHO-1}, timeoutQC \rangle_*$
  $v_h \leftarrow$ the view number of the highest $timeoutQC$ among received $\langle \text{ECHO-1} \rangle$ messages
  wait for a $timeoutQC$ tc such that $view(tc) \geq v_h + 2$
  $timeoutQC \leftarrow tc$
  broadcast $\langle \text{RECOVERY-2}, tc \rangle_i$
  wait for $n - f - s$ $\langle \text{ECHO-2}, v', (pqc, lqc, cqc) \rangle_*$ where: pqc is a $prepareQC$, lqc is a $lockedQC$, cqc is a $commitQC$, and $v' > v_h + 2$
  $lockedQC \leftarrow$ the $lockedQC$ with the highest view number among received $\langle \text{ECHO-2} \rangle$ messages
  $prepareQC \leftarrow$ the $prepareQC$ with the highest view number among received $\langle \text{ECHO-2} \rangle$ messages
  $commitQC \leftarrow$the $commitQC$ with the highest view number among received $\langle \text{ECHO-2} \rangle$ messages
  deliver the block corresponding to $commitQC$ and all its ancestors
  $cv \leftarrow view(timeoutQC) + 1$
  send $\langle \text{NEW-VIEW}, cv, prepareQC \rangle_i$ to the leader of cv
  set status to *awake* and rejoin the main protocol's execution
as an awake replica
  upon receiving $\langle \text{RECOVERY-1} \rangle_j$ **do**
    send $\langle \text{ECHO-1}, timeoutQC \rangle_i$ to p_j
  upon receiving $\langle \text{RECOVERY-2}, timeoutQC \rangle_j$ **do**
    if $view(timeoutQC) \geq cv$ **then**
      start view change and enter $view(timeoutQC) + 1$
    send $\langle \text{ECHO-2}, cv, (pqc, lqc, cqc) \rangle_i$ to p_j
    forward to p_j all delivered blocks, the block B corresponding to $prepareQC$, and all ancestors of B

recovery protocol. Suppose that an honest replica p_i fell asleep in view v. During recovery, p_i must obtain a $timeoutQC$ with a view number of at least $v - 2$, i.e., if the highest received $timeoutQC$ is formed in view v_h, $v_h \geq v - 2$. According to the recovery protocol, p_i waits for a timeoutQC tc with view number of at least $v_h + 2$ and then enter view $view(tc) + 1$, i.e., it enters view $v_h + 3 \geq v + 1$. Thus, p_i will not vote twice for blocks in the same view, so safety is achieved. Meanwhile, liveness roughly follows that of HotStuff, as we only modify the quorum size. The recovery protocol is non-blocking, as a recovering replica can complete the protocol and obtain a $prepareQC$ no lower than its $lockedQC$ before it fell asleep.

Koala-2 achieves $O(\kappa n^2)$ communication, where κ is the security parameter. The recovery protocol achieves $O(\kappa n + lnL)$ communication, where L is the size of a block and l is the length of the longest chain corresponding to a $prepareQC$.

VI. Implementation and Evaluation

We implement Koala-1, MMR [12], HotStuff [7], and Koala-2 in Golang[2]. Our implementation involves around 8,500 LOC for the protocols and about 1,000 LOC for evaluation. We use gRPC as the communication library. We use ECDSA to realize the authenticated channel and use SHA256 as the underlying hash function. We use LevelDB[3] to implement the stable storage.

We evaluate the performance of our protocols on Amazon EC2 using up to 91 virtual machines (VMs). We use *m5.xlarge* instances for our evaluation. The m5.xlarge instance has four virtual CPUs and 16GB memory. We evaluate protocols in both LAN and WAN. Unless otherwise mentioned, we report the results in the WAN setting, where replicas are evenly distributed in up to six regions: us-west-2 (Oregon, US), us-east-2 (Ohio, US), ap-southeast-2 (Sydney, Australia), ap-northeast-1 (Tokyo, Japan), sa-east-1 (São Paulo, Brazil), and eu-west-1 (Ireland).

We conduct experiments under different network sizes and batch sizes. We use f to denote the network size. For Koala-1 and MMR, we use $n = 2f + 1$ replicas in total, assuming that all replicas remain awake. For HotStuff, we use $n = 3f + 1$ replicas. For Koala-2, we use $n = 3f + 2s + 1$ replicas, and we vary s to report the performance. We use b to denote the batch size, where a leader proposes b transactions in each block. The default transaction size is 250 bytes.

TABLE III: **The latencies of Koala-1 and MMR.**

Protocol	The number of Byzantine replicas	Expected latency (ms)	Best-case latency (ms)
Koala-1	$f = 10$	5019.70	2117.80
	$f = 20$	5272.74	2400.75
MMR [12]	$f = 10$	11194.71	4000.00
	$f = 20$	13668.65	3999.96

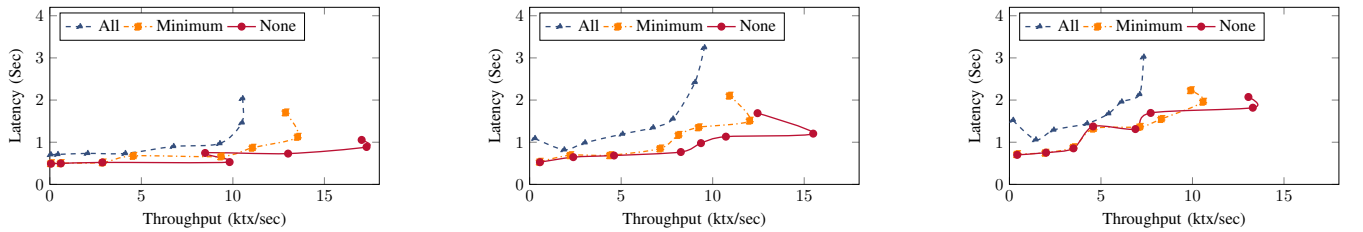
Koala-1 vs. MMR. We evaluate the best-case latency and expected latency of Koala-1 and MMR for $f = 10$ and $f = 20$. As both protocols assume a synchronous network, we evaluate them in the LAN setting, where all instances are launched in the us-west-2 (Oregon, US) region. We set Δ as one second. For each experiment, we report the average latency of *a-delivering* 200 blocks. As shown in Table III, the latency of MMR is consistently higher than that of Koala-1. Specifically, the best-case latency of Koala-1 is about 56.5% that of MMR and the expected latency of Koala-1 is about 41.7% that of MMR. This matches with our theoretical results in Table II.

HotStuff under different storage options. We evaluate three modes to support our result in Sec. IV: *all* consensus parameters are stored in stable storage (the default solution to build sleepy consensus with $n = 3f + 1$ setting); *minimum* parameters are stored (i.e., HotStuff-mSS); *no* intermediate parameters are stored (sleepy consensus cannot be achieved in the $n = 3f + 1$ setting).

We show the latency vs. throughput for $f = 10, 20, 30$ in Figures 4a, 4b, and 4c. For each f, the performance is the highest if no consensus parameters are stored in stable storage and the lowest if all consensus parameters are stored in stable storage. This is expected as stable storage involves

[2]Part of our codebase can be found at: https://github.com/Spongebob-bear/Koala-NDSS-AE or https://doi.org/10.5281/zenodo.16956543

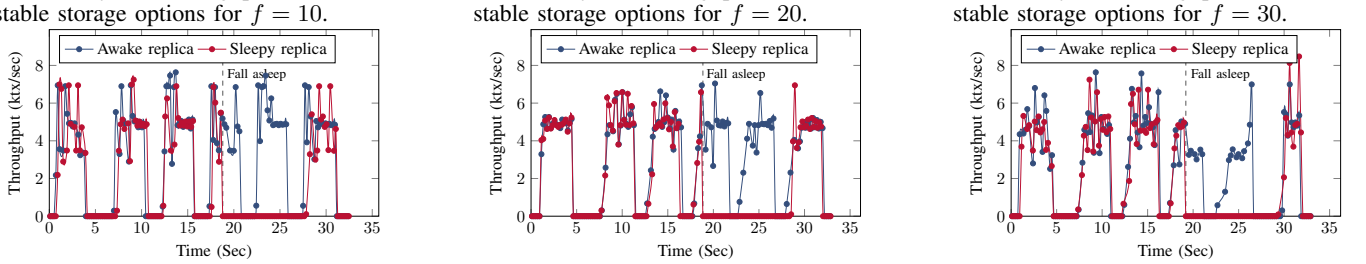
[3]https://github.com/syndtr/goleveldb



(a) Latency vs. throughput of different stable storage options for $f = 10$.

(b) Latency vs. throughput of different stable storage options for $f = 20$.

(c) Latency vs. throughput of different stable storage options for $f = 30$.



(d) Throughput of Koala-2 for $s = 10$.

(e) Throughput of Koala-2 for $s = 20$.

(f) Throughput of Koala-2 for $s = 30$.

Fig. 4: The performance of HotStuff under different storage options (Sec. IV) and the performance of Koala-2 (Sec. V) with different numbers of sleeping replicas.

hard disk I/O. In contrast, the performance of HotStuff-mSS is already very close to that of storing no consensus parameters in stable storage, especially when f is large. When $f = 30$, the peak throughput of HotStuff-mSS is 44.3% higher than that of storing all consensus parameters and 20.4% lower than that of storing no consensus parameters.

Performance of Koala-2. We show the throughput of Koala-2 for $s = 10, 20, 30$ in Figures 4d, 4e, and 4f, respectively. View change is started every 5 seconds. We make s replicas fall asleep in the middle of the experiment and immediately start the recovery procedure afterward.

Our results show that our recovery procedure introduces short-lived throughput fluctuation. When the recovery procedure begins, the throughput of an awake replica degrades to at most 53.4% of the average throughput. This is because replicas need to synchronize their latest $timeoutQC$ with the recovering replicas. However, the degradation only lasts for a short period as our recovery procedure only involves four communication steps. Moreover, the throughput fluctuation does not become more obvious as s grows. This result shows that our recovery mechanism is efficient.

We also show the latency of the recovery protocol in Figures 4d, 4e, and 4f. The recovery time is 9.25s for $s = 10$, 9.89s for $s = 20$, and 10.38s for $s = 30$. This is expected since the recovery protocol requires a recovering replica to wait for two subsequent view changes.

Koala-2 vs. HotStuff without stable storage. We assess the throughput of Koala-2 and HotStuff without stable storage and show the results in Figure 5. We focus on the failure-free case, since HotStuff without stable storage is not safe or live in other cases. We fix $f = 10$ and vary the number of sleepy replicas s. For HotStuff, the total number of replicas is fixed at $n = 3f + 1 = 31$. For Koala-2, the number of replicas is $n = 3f + 2s + 1$. Our results show that when $s = 0$, the throughput of both protocols is nearly identical. As s increases, Koala-2's throughput decreases. This is because Koala-2 requires a larger number of replicas when s increases, leading to a higher communication overhead.

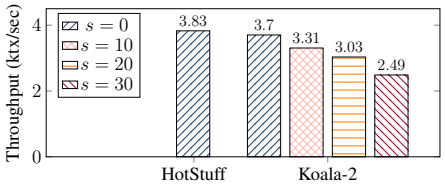


Fig. 5: The performance comparison between Koala-2 and HotStuff without stable storage.

Demo of double spending attack. Our attack in Sec. IV-A can be adapted into a *double-spending attack* against decentralized payment systems that use partially synchronous BFT consensus. We implement a demo of a decentralized payment system to simulate the double-spending attack. We configure the consensus layer using three different protocols: HotStuff without stable storage, HotStuff with minimum consensus parameters stored (i.e., HotStuff-mSS), and Koala-2.

We use four replicas when running HotStuff and six replicas when running Koala-2. In both cases, replica 0 is designated as the leader of view 0 and is Byzantine. Also, an honest replica falls asleep upon receiving the fifth block and wakes up after a few seconds. When running HotStuff without stable storage, as shown in Figure 6, the sleepy replica delivers a new block after waking up, and the block consists of a transaction that causes double spending. This validates the result in Sec. IV-A. When running HotStuff-mSS and Koala-2, as shown in Figure 7 and

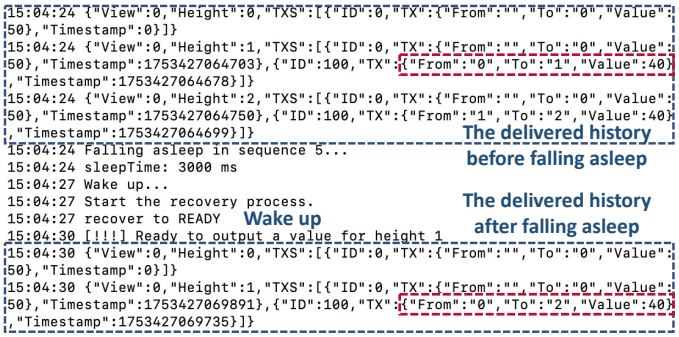


Fig. 6: The double-spending attack to HotStuff without stable storage. Before sleep, an honest replica delivers a block at height 1, containing a transaction where replica 0 transfers 40 tokens to replica 1. After the replica falls asleep and later wakes up, it delivers a new block at height 1, containing a transaction where replica 0 transfers 40 tokens to replica 2. In this way, a double-spending attack is performed successfully.

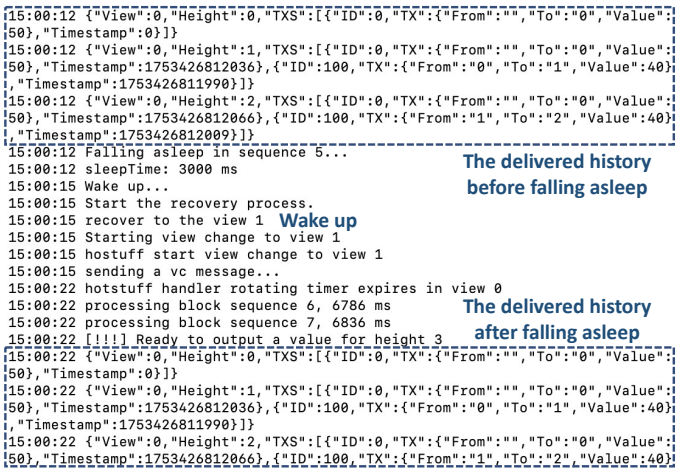


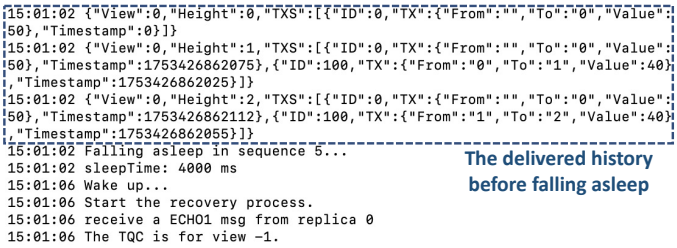
Fig. 7: HotStuff-mSS defends against the double-spending attack. When the sleepy replica recovers, it restores its $lockedQC$ and view number from stable storage. After waking up, it delivers blocks that extend the block before it falls asleep. Thus, the double-spending attack fails.

Figure 8, no double spending is caused. This is expected by our analysis in Sec. IV-B and Sec. V.

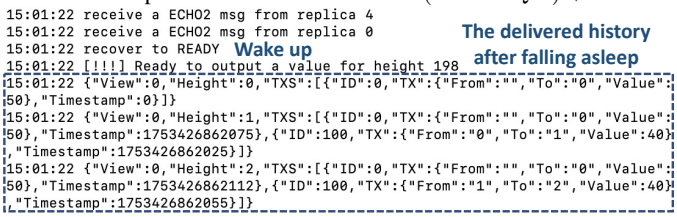
VII. Additional Related Work

Synchronous BFT and sleepy consensus. Under the unknown participation model, sleepy consensus can only be achieved in a synchronous network [9]. Many recent sleepy consensus focus on lowering the latency [10], [12]. A recent work by D'Amato et al. [13] achieves lower latency than MMR, but a new stable participation assumption is introduced. Goldfish [40] achieves 4Δ latency under a high participation level (i.e., h_a/n is high), but exhibits much longer latency than MR in the worst case.

There are some variants of sleepy consensus. Gafni and Losa [11] studied Byzantine agreement in the sleepy model



(a) When the sleepy replica recovers, it collects the latest $timeoutQC$ from awake replicas and enters view $view(timeoutQC) + 3$.



(b) During recovery, the replica collects the latest $lockedQC$ from awake replicas. After waking up, it delivers blocks that extend the block before it falls asleep. The double-spending attack fails.

Fig. 8: Koala-2 defends against the double-spending attack.

that achieves constant latency. A recent work by D'Amato, Losa, and Zanolini [41] studies asynchrony resilience for synchronous sleepy consensus. The idea is to make a synchronous protocol safe and live under intermittent asynchronous periods.

Our protocols can be viewed as BFT that also tolerates sleepy replicas. Different from sleepy consensus, we assume a known h_a. The assumptions we made about the known h_a are stronger than sleepy consensus. Namely, sleepy consensus only assumes that at each point, over half of the awake replicas are honest, but the number of awake honest replicas is unknown. Our requirement on h_a might require a larger number of awake honest replicas compared to sleepy consensus. Although our assumptions are stronger, they are realistic and useful in practice. Indeed, in most industrial systems, most replicas are awake. For instance, the participation rate of Ethereum is 99.8%[4] out of over a million replicas.

Variants of consensus in the unknown participation model. There are some variants of consensus where n is unknown [42]–[44]. In these works, each replica is not directly connected to all replicas in the system when the system starts. These works do not handle sleepy replicas.

Dynamic BFT. Dyno [34] studies dynamic BFT in the partially synchronous model, where replicas may join and leave the system. Khanchandani and Wattenhofer [45] study Byzantine agreement when both n and f are unknown and replicas can join and leave the system. MITOSIS [46] provides a dynamic sharding approach. This line of work assumes that honest replicas that still participate in the system are always awake. In contrast, we assume sleepy replicas where honest replicas in the system may go asleep.

[4]Data source (accessed in Jul 2025): https://www.rated.network/

Sleepy replicas vs. crash-recovery model. Before MR, all sleepy consensus protocols assumed *message delivery*, as defined in Sec. III. MR pointed out that such an assumption might be strong. As a solution to remove the assumption, a *recovery* protocol is proposed for replicas that fall asleep to catch up with awake replicas. Accordingly, all sleepy consensus protocols known so far without the message delivery assumption can be considered protocols that tolerate both Byzantine failures and crash-recovery failures [10], [12], [13].

Consensus in the mixed failure model. Several work study protocols with both Byzantine failures and crash failures [18]–[21]. In the partially synchronous network, UpRight [20] implements a BFT protocol assuming $n \geq 3f + 2c + 1$, where c is the number of crash failures. Scrooge [18] presents a fast Byzantine agreement protocol that requires $n \geq 4f + 2c$. SBFT [19] provides a BFT-SMR protocol assuming $n \geq 3f + 2c + 1$. In the synchronous network, a recent work [21] proves a lower bound of $n \geq 2f + c + 1$. In contrast, our model considers both Byzantine failures and sleepy replicas. As replicas may lose their state after they become asleep, the model we consider is more challenging compared to prior works. The bound we show in the partially synchronous model without GAT is $n \geq 3f + 2s + 1$, which resembles that with Byzantine and crash failures.

Diskless crash recovery. Consensus in the crash-recovery model has been studied for crash fault-tolerant protocols [47]–[49]. Most protocols rely on the stable storage assumption. Protocols without the stable storage assumption are also known as protocols in the diskless crash recovery (DCR) model [50]. Aguilera, Chen, and Toueg [51] discuss under what conditions stable storage is necessary. Michael, Ports, Sharma, and Szekeres [50] provide a generic approach that transforms protocols in the crash-recovery model (with stable storage) to the DCR model. All these works consider benign crash failures. In contrast, our Koala-2 protocol can be considered as the first BFT protocol in the DCR model.

VIII. Conclusion

We study consensus with Byzantine failures and sleepy replicas in the known participation model, where all awake replicas are aware of the minimum number of awake honest replicas. Such a model has practical implications for systems where honest parties might crash and later recover. We provide three results in both synchronous and partially synchronous networks.

Acknowledgment

This study was supported by the National Key R&D Program of China (No.2023YFB2703600), the National Natural Science Foundation of China (No. 62302266, 62232010, U23A20302, U24A20244), the Shandong Science Fund for Excellent Young Scholars (No.2023HWYQ-008), and the project ZR2022ZD02 supported by Shandong Provincial Natural Science Foundation. This study was also supported in part by the National Natural Science Foundation of China under 92267203, Beijing Natural Science Foundation under M23015.

Ethics Considerations

This research is committed to the principles of research ethics. In particular, we adhere to the following principles:

- **Respect for persons**: No personal data was used in our research. The transaction data in the demo of the double-spending attack was fabricated, solely to demonstrate the double-spending attack.
- **Beneficence**: Our research offers a fine-grained treatment of consensus that tolerates both Byzantine failures and sleepy replicas, which can improve existing blockchain systems and potentially mitigate the risk of economic loss for investors.
- **Justice**: The security of our protocols has been proven. Therefore, all developers who implement our protocols in their blockchain systems can safeguard their systems against sleepy replicas, ensuring that every user's benefits are equally protected as long as the protocol is followed.
- **Respect for law and public interest**: We ensure that our research complies with all relevant laws and regulations. Our methods and results are transparent, with no aspects concealed that might violate existing laws.

References

[1] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *CCS*, 2016, pp. 31–42.
[2] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *CCS*, 2018, pp. 2028–2041.
[3] H. Zhang, S. Duan, B. Zhao, and L. Zhu, "WaterBear: Practical asynchronous BFT matching security guarantees of partially synchronous BFT," in *USENIX Security*, 2023, pp. 5341–5357.
[4] H. Zhang, C. Liu, and S. Duan, "How to achieve adaptive security for asynchronous bft?" *JPDC*, vol. 169, pp. 252–268, 2022.
[5] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999, pp. 173–186.
[6] S. Duan, X. Wang, and H. Zhang, "Practical signature-free asynchronous common subset in constant time," in *CCS*, 2023, pp. 815–829.
[7] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *PODC*, 2019, pp. 347–356.
[8] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[9] R. Pass and E. Shi, "The sleepy model of consensus," in *ASIACRYPT*, 2017, pp. 380–409.
[10] A. Momose and L. Ren, "Constant latency in sleepy consensus," in *CCS*, 2022, pp. 2295–2308.
[11] G. Losa and E. Gafni, "Consensus in the unknown-participation message-adversary model," *arXiv preprint arXiv:2301.04817*, 2023.
[12] D. Malkhi, A. Momose, and L. Ren, "Towards practical sleepy bft," in *CCS*, 2023, pp. 490–503.
[13] F. D'Amato, R. Saltini, T.-H. Tran, and L. Zanolini, "Tob-svd: Total-order broadcast with single-vote decisions in the sleepy model," *arXiv preprint arXiv:2310.11331*, 2024.
[14] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas, "Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability," in *CCS*, 2018, pp. 913–930.
[15] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas, "Ouroboros chronos: Permissionless clock synchronization via proof-of-stake," *Cryptology ePrint Archive*, 2019.
[16] E. Budish, A. Lewis-Pye, and T. Roughgarden, "The economic limits of permissionless consensus," in *EC*, 2024, pp. 704–731.
[17] J. Neu, E. N. Tas, and D. Tse, "Ebb-and-flow protocols: A resolution of the availability-finality dilemma," in *SP*, 2021, pp. 446–465.
[18] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri, "Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas," in *DSN*, 2010, pp. 353–362.
[19] 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 *DSN*, 2019, pp. 568–580.

[20] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *SOSP*, 2009, pp. 277–290.
[21] I. Abraham, D. Dolev, A. Kagan, and G. Stern, "Authenticated consensus in synchronous systems with mixed faults," *Cryptology ePrint Archive*, 2022.
[22] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.
[23] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.
[24] M. Zhang, R. Li, and S. Duan, "Max attestation matters: Making honest parties lose their incentives in ethereum pos," in *USENIX Security*, 2024.
[25] Diem Association, "The diem blockchain," 2020, available at: https://developers.diem.com/docs/technical-papers/the-diem-blockchain-paper/. Accessed in 9-2024.
[26] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," *arXiv preprint arXiv:1807.04938*, 2019.
[27] Diem Association, "Diem BFT project page," 2020, available at: https://github.com/diem/diem. Accessed in 9-2024.
[28] M. Davis and H. Vandierendonck, "Achieving scalable consensus by being less writey," in *HPDC*, 2021, pp. 257–258.
[29] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *USENIX ATC*, 2013, pp. 169–180.
[30] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Byzantine agreement, broadcast and state machine replication with near-optimal good-case latency," *arXiv preprint arXiv:2003.13155*, 2020.
[31] J. Neu, E. N. Tas, and D. Tse, "The availability-accountability dilemma and its resolution via accountability gadgets," in *FC*, 2022, pp. 541–559.
[32] B. Y. Chan and E. Shi, "Streamlet: Textbook streamlined blockchains," in *AFT*, 2020, p. 1–11.
[33] J. Sousa, E. Alchieri, and A. Bessani, "State machine replication for the masses with BFT-SMaRt," in *DSN*, 2014, pp. 355–362.
[34] S. Duan and H. Zhang, "Foundations of dynamic bft," in *SP*, 2022, pp. 1317–1334.
[35] X. Sui, S. Duan, and H. Zhang, "Marlin: Two-phase bft with linearity," in *DSN*, 2022, pp. 54–66.
[36] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *JACM*, vol. 35, no. 2, pp. 288–323, 1988.
[37] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *SP*, 2020, pp. 106–118.
[38] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected o (1) rounds, expected communication, and optimal resilience," in *FC*. Springer, 2019, pp. 320–334.
[39] C. Wang, S. Duan, M. Xu, F. Li, and X. Cheng, "Consensus in the known participation model with byzantine failures and sleepy replicas," Cryptology ePrint Archive, 2024. [Online]. Available: https://eprint.iacr.org/2024/137
[40] F. D'Amato, J. Neu, E. N. Tas, and D. Tse, "Goldfish: No more attacks on ethereum?!" in *FC*, 2025, p. 3–23.
[41] F. D'Amato, G. Losa, and L. Zanolini, "Asynchrony-resilient sleepy total-order broadcast protocols," in *PODC*, 2024, p. 247–256.
[42] H. Heydari, R. Vassantlal, and A. Bessani, "Knowledge connectivity requirements for solving bft consensus with unknown participants and fault threshold," in *ICDCS*, 2024, pp. 221–231.
[43] E. A. P. Alchieri, A. Bessani, F. Greve, and J. d. S. Fraga, "Knowledge connectivity requirements for solving byzantine consensus with unknown participants," *TDSC*, vol. 15, no. 2, pp. 246–259, 2018.
[44] E. Alchieri, A. Bessani, J. Silva Fragao, and F. Greve, "Byzantine consensus with unknown participants," in *OPODIS*, 2008.
[45] P. Khanchandani and R. Wattenhofer, "Byzantine agreement with unknown participants and failures," in *IPDPS*. IEEE, 2021, pp. 952–961.
[46] G. A. Marson, S. Andreina, L. Alluminio, K. Munichev, and G. Karame, "Mitosis: practically scaling permissioned blockchains," in *ACSAC*, 2021, pp. 773–783.
[47] E. Jiménez, J. L. López-Presa, and M. Patiño-Martínez, "Consensus in anonymous asynchronous systems with crash-recovery and omission failures," *Computing*, vol. 103, no. 12, pp. 2811–2837, 2021.
[48] M. Hurfin, A. Mostefaoui, and M. Raynal, "Consensus in asynchronous systems where processes can crash and recover," in *SRDS*, 1998, pp. 280–286.
[49] M. Backes and C. Cachin, "Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries," in *DSN*, vol. 3, 2003, pp. 37–46.
[50] E. Michael, D. R. Ports, N. K. Sharma, and A. Szekeres, "Recovering shared objects without stable storage," in *DISC*, 2017, pp. 36:1–36:16.
[51] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," *Distributed computing*, vol. 13, pp. 99–125, 2000.

APPENDIX A
PROOF OF KOALA-1

Proof of our VT-GPE. We begin with the correctness of our VT-GPE protocol shown in Algorithm 1. We split the graded delivery property into graded delivery-1 and graded delivery-2 for our proof.

Lemma 1 (External Validity)**.** *If an honest replica tgpe-decides* $(B, *, *)$ *such that* $B \neq \perp$*, at least one honest replica has verified* B *and* $Q(B, \sigma)$ *holds at it, where* σ *is the proof of* B*.*

Proof. If an honest replica p_i *tgpe-decides* $(B, *, *)$, p_i holds valid $E(B)$, i.e., p_i receives $f+1$ matching $\langle \text{ECHO} \rangle$ messages for B. At least one of the $\langle \text{ECHO} \rangle$ messages is from an honest replica. This replica must have verified B before echoing B, thus $Q(B, \sigma)$ holds. □

Lemma 2 (Consistency)**.** *If an honest replica tgpe-decides* $(B, *, *)$ *and another honest replica tgpe-decides* $(B', *, *)$*,* $B = B'$*.*

Proof. Assuming that p_i *tgpe-decides* $(B, *, *)$ and p_j *tgpe-decides* $(B', *, *)$ and $B \neq B'$. According to Lemma 1, $Q(B, *)$ holds for at least one honest replica p_1 and $Q(B', *)$ holds for at least one honest replica p_2. In this case, p_1 must have sent an $\langle \text{ECHO} \rangle$ message for B at $t = \Delta$ and p_2 must have sent an $\langle \text{ECHO} \rangle$ message for B' at $t = \Delta$. As each honest replica sends an $\langle \text{ECHO} \rangle$ message for block B only if B is a winning input, p_1 must have forwarded B at $t = \Delta$. Similarly, p_2 has forwarded B' at $t = \Delta$. Therefore, every honest replica must have received $\langle \text{INPUT} \rangle$ messages for both B and B' by $t = 2\Delta$. At most one of these two inputs could be chosen as the winning input by any honest replica at $t = 2\Delta$.

Suppose the B is chosen by all honest replicas after $t = 2\Delta$. No honest replicas will send $\langle \text{WINNER1} \rangle$ or $\langle \text{READY} \rangle$ messages for B' at $t = 2\Delta$. No honest replicas will send $\langle \text{LOCK} \rangle$ messages for B' at $t = 3\Delta$. Since replicas need to receive $\langle \text{WINNER1} \rangle$, $\langle \text{READY} \rangle$, or $\langle \text{LOCK} \rangle$ messages from at least one honest replica to *tgpe-decide* block B' with grade 0, 1, and 2, none of them would *tgpe-decide* B', a contradiction. □

Corollary 1. *If an honest replica receives a valid* $E(B)$ *and* $W_1(B)$ *for a block* B *and another honest replica receives a valid* $E(B')$ *and* $W_1(B')$ *for a block* B' *with* $view(B) = view(B')$*,* $B = B'$*.*

Proof. Suppose $view(B) = view(B') = v$. According to the protocol, an honest replica p_i will *tgpe-decide* B in $\mathsf{VT\text{-}GPE}_v$ when it receives a valid $E(B)$ and $W_1(B)$ for B. Similarly, another honest replica p_j will *tgpe-decide* B' in $\mathsf{VT\text{-}GPE}_v$

when it receives a valid $E(B')$ and $W_1(B')$ for B'. Due to Lemma 2, $B = B'$. □

Lemma 3 (Graded Delivery-1)**.** *If an honest replica tgpe-decides* $(B, 1, *)$*, any honest replica tgpe-decides* $(B, 0, *)$*.*

Proof. If an honest replica p_1 *tgpe-decides* $(B, 1, *)$, it must have received at least $f + 1$ valid $\langle \text{READY} \rangle$ messages for B and at least one honest replica p_2 has broadcast the $\langle \text{READY} \rangle$ message for B at $t = 2\Delta$. Before p_2 sent the $\langle \text{READY} \rangle$ message, it must have collected a valid $E(B)$ (i.e., $f + 1$ $\langle \text{ECHO} \rangle$ messages) and forwarded $E(B)$. Therefore, at $t \geq 3\Delta$, every honest replica can collect a valid $E(B)$.

Let the proposer of B be p_3. Below we prove that all honest replicas must have observed a winning input for B at time $t = 2\Delta$. Firstly, according to the protocol, replica p_1 must have received at least $f+1$ valid $\langle \text{WINNER2} \rangle$ messages for B when it *tgpe-decides* B. In this case, an honest replica must have observed a winning input for B at $t = 3\Delta$. Therefore, at $t = 2\Delta$, no honest replica could observe a VRF evaluation higher than the VRF evaluation generated by p_3. Furthermore, no equivocation by p_3 is detected. Meanwhile, all honest replicas must have received the $\langle \text{INPUT} \rangle$ for B by $t = 2\Delta$. This is because p_2 already has $|E(B)| \geq f + 1$ by $t = 2\Delta$ so at least one honest replica has previously set the $\langle \text{INPUT} \rangle$ message for B as its winning input by $t = \Delta$. As the honest replica forwards the $\langle \text{INPUT} \rangle$ message, any honest replicas awake at $t = 2\Delta$ must have considered the $\langle \text{INPUT} \rangle$ message for B as their winning input and sent $\langle \text{WINNER1} \rangle$ messages for B.

Since at least $h_a = f + 1$ honest replicas are awake at $t = 2\Delta$, any honest replicas awake at time $t \geq 4\Delta$ must have $|E(B)| \geq f + 1$ and $|W_1(B)| \geq f + 1$ and then *tgpe-decide* $(B, 0, *)$. □

Lemma 4 (Graded Delivery-2)**.** *If an honest replica tgpe-decides* $(B, 2, *)$*, any honest replica tgpe-decides* $(B, 1, *)$*.*

Proof. If an honest replica p_1 *tgpe-decides* $(B, 2, *)$, it must have received at least $f+1$ valid $\langle \text{LOCK} \rangle$ messages for B and at least one honest replica p_2 has sent a $\langle \text{LOCK} \rangle$ message for B at $t = 3\Delta$. Before p_2 sent the $\langle \text{LOCK} \rangle$ message, it must have collected a valid $R(B)$ (at least $f+1$ matching $\langle \text{READY} \rangle$ messages) and forwarded $R(B)$. Therefore, at $t \geq 4\Delta$, every honest replica can collect a valid $R(B)$.

Let the proposer of B be p_3. Below we prove that all honest replicas must have observed a winning input for B at time $t = 3\Delta$. Firstly, when p_1 *tgpe-decides* B, it must have observed a winning input for B at $t = 4\Delta$. Therefore, at $t = 3\Delta$, no honest replica could observe a VRF evaluation higher than that of p_3 or any equivocating messages by p_3. Meanwhile, all honest replicas must have received the $\langle \text{INPUT} \rangle$ message for B by $t = 3\Delta$. This is because p_2 has $|R(B)| \geq f + 1$ at time $t = 3\Delta$ and at least one honest replica has previously sent a $\langle \text{READY} \rangle$ message at time $t = 2\Delta$. The honest replica must have forwarded the $\langle \text{INPUT} \rangle$ message for B at $t = 2\Delta$. As a result, all honest replicas awake at $t = 3\Delta$ must have considered the $\langle \text{INPUT} \rangle$ message for B as their winning input and sent $\langle \text{WINNER2} \rangle$ messages for B.

Since at least $h_a = f + 1$ honest replicas are awake at $t = 3\Delta$, any honest replicas awake at any $t \geq 4\Delta$ have $|R(B)| \geq f+1$ and $|W_2(B)| \geq f+1$ and then *tgpe-decide* $(B, 1, *)$. □

Lemma 5 (Validity)**.** *With a probability of* $\alpha > 1/2$*, all honest replicas tgpe-decide* $(B, 2, *)$ *where block* B *is tgpe-proposed by an honest replica.*

Proof. As at least $h_a = f + 1$ honest replicas are awake at time $t = 0$ and there are at most f faulty replicas, with probability $\alpha > 1/2$, an honest replica's VRF evaluation will be the highest among all awake replicas. Let the replica be p_1 and the block p_1 *tgpe-proposes* be (B, σ), where σ is the proof of block B. After p_1 broadcasts its $\langle \text{INPUT} \rangle$ message, all honest replicas awake at time $t \geq \Delta$ will set their winning input as the $\langle \text{INPUT} \rangle$ message for B.

As p_1 is an honest replica, $Q(B, \sigma)$ holds at all honest replicas. It is then not difficult to see that any honest replica broadcasts a $\langle \text{ECHO} \rangle$ message for B at $t = \Delta$. Each honest replica awake at $t = 2\Delta$ observes a valid $E(B)$ such that $|E(B)| \geq f + 1$ and broadcasts a $\langle \text{WINNER1} \rangle$ and a $\langle \text{READY} \rangle$ message for B. Similarly, all honest replicas awake at $t = 3\Delta$ observe a valid $R(B)$ such that $|R(B)| \geq f + 1$. Therefore, they broadcast $\langle \text{WINNER2} \rangle$ and $\langle \text{LOCK} \rangle$ messages for B. Finally, at $t \geq 4\Delta$, all awake honest replicas will observe a valid $L(B)$ such that $|L(B)| \geq f + 1$ and then *tgpe-decide* $(B, 2, *)$. □

Proof of Koala-1. We now prove the correctness of our ABC protocol. In this section, we prove the correctness of the protocol shown in Algorithm 2 (the none-pipelining mode).

Theorem 2 (Safety)**.** *If an honest replica a-delivers a block* B_1 *before it a-delivers a block* B_2*, then no honest replica a-delivers the block* B_2 *without first a-delivering* B_1*.*

Proof. Suppose an honest replica p_1 *a-delivers* block B_1 before it *a-delivers* B_2 and another honest replica p_2 *a-delivers* B_2 before it *a-delivers* B_1. W.l.o.g., we assume that p_1 *a-delivers* B_1 after it *tgpe-decides* $(B_1, 2, *)$ in $\mathsf{VT\text{-}GPE}_{v_1}$. Additionally, p_2 *a-delivers* B_2 after it *tgpe-decides* $(B_2, 2, *)$ in $\mathsf{VT\text{-}GPE}_{v_2}$. Obviously, $v_1 \neq v_2$, as otherwise the consistency property of VT-GPE is violated. W.l.o.g, let $v_1 < v_2$.

According to Lemma 4, if p_1 *tgpe-decides* $(B_1, 2, *)$ for block B_1 in $\mathsf{VT\text{-}GPE}_{v_1}$, any honest replica p_i (including p_2) *tgpe-decides* $(B_1, 1, *)$ in $\mathsf{VT\text{-}GPE}_{v_1}$. Furthermore, if p_i *tgpe-decides* $(B_1, 1, *)$ in $\mathsf{VT\text{-}GPE}_{v_1}$, by Lemma 3, any honest replica will *tgpe-decide* $(B_1, 0, qc_1)$ in $\mathsf{VT\text{-}GPE}_{v_1}$. According to our protocol, qc_1 is a valid $prepareQC$ with $f + 1$ $\langle \text{ECHO} \rangle$ messages and $f + 1$ $\langle \text{WINNER1} \rangle$ messages for B_1. Therefore, any honest replica that enters the next view $v_1 + 1$ uses qc_1 as input. Furthermore, since any honest replica (including p_2) *tgpe-decides* $(B_1, 1, *)$, the replica sets its lock as B_1. The lock parameter can be set as a block that extends B_1 unless the replica becomes unlocked on B_1.

Since p_2 *tgpe-decides* $(B_2, 2, *)$ in view v_2 and is locked on B_1 in view v_1 (where $v_1 < v_2$), there must exist a view

v_3 such that the following holds: 1) $v_1 < v_3 \leq v_2$; 2) an honest replica *tgpe-decides* a block B_3 in VT-GPE$_{v_3}$ and B_3 is conflicting with B_1; 3) a valid qc_3 is provided by the proposer of block B_3 and $Q(B_3, qc_3)$ is verified by at least one honest replica (as otherwise the external validity property of VT-GPE is violated). Here, $view(qc_3) < v_3$ as qc_3 is a proof included in the proposal of block B_3. W.l.o.g., suppose v_3 is the first view such that the above holds.

Towards a contradiction, we now show that B_3 cannot be a conflicting block of B_1. According to our protocol, qc_3 is a $prepareQC$ and consists of $f+1$ matching $\langle \text{ECHO} \rangle$ and $f+1$ matching $\langle \text{WINNER1} \rangle$ messages. Any honest replica p_k that verifies $Q(B_3, qc_3)$ in view v_3 must have a lock (denoted as lock$_k$) such that $view(\text{lock}_k) \leq view(qc_3)$. As $view(\text{lock}_k) \geq v_1$, now there are two cases: $view(qc_3) = v_1$ and $view(qc_3) > v_1$. If $view(qc_3) = v_1$, qc_3 and qc_1 must have been formed in VT-GPE$_{v_1}$, where qc_1 is a valid $prepareQC$ for B_1. Both qc_3 and qc_1 have been received by any honest replica awake after view v_1. According to Corollary 1, the block for qc_3 is B_1, a contradiction. If $view(qc_3) > v_1$, we have $v_1 < view(qc_3) < v_3 \leq v_2$. The block corresponding to qc_3 is a conflicting block with B_1 and has been verified by at least one honest replica. However, we already assume that v_3 is the first view such that a conflicting block is proposed, a contradiction.

As B_3 cannot be a conflicting block of B_1, block B_2 extends block B_1. However, p_2 *a-delivers* B_1 after it *a-delivers* B_2, a contradiction. □

Theorem 3 (Liveness)**.** *If an honest replica a-broadcasts a message m, then all awake honest replicas eventually a-deliver m.*

Proof. We first prove that any block (B_1, qc) *tgpe-proposed* by any honest replica p_1 in a view v_1 can be verified by all honest replicas such that $Q(B_1, qc)$ holds. At the beginning of view v_1, B_1 extends the candidate of p_1 and qc is a $prepareQC$ of candidate. As p_1 broadcasts an $\langle \text{INPUT} \rangle$ message for (B_1, qc) in VT-GPE$_{v_1}$, all awake honest replicas eventually receive the $\langle \text{INPUT} \rangle$ message for B_1. According to the graded delivery-1 property of VT-GPE, in any VT-GPE$_v$ such that $v < v_1$, if any honest replica *tgpe-decides* a block B with grade 1, p_1 must have *tgpe-decided* $(B, 0, *)$ and set its candidate as B. Therefore, the view number of p_1's candidate must be equal to or higher than that of the lock of any honest replica in view v_1. $Q(B_1, qc)$ thus holds at any honest replica.

According to the validity property of VT-GPE, with a probability of $\alpha > 1/2$, all honest replicas will *tgpe-decide* $(B, 2, *)$ for a block B in a VT-GPE instance. With trivial input dissemination, honest replicas can broadcast their *a-broadcast* messages and any honest replica can *a-broadcast* the messages that have not been *a-delivered*. It is then not difficult to see that any message m *a-broadcast* by an honest replica will eventually be *a-delivered* within a constant number of views. □

APPENDIX B
PRACTICAL RECOVERY PROTOCOL FOR KOALA-1

In Koala-1 presented in Sec. III, we have assumed stable storage and message delivery. In this section, we provide a practical recovery protocol to remove this assumption. Similar to prior works [10], [12], the recovery protocol is used for *recovering* replicas (that become awake after sleeping) to catch up with awake replicas.

We follow the notations used by prior works and define a third status (besides awake and asleep) called *recovering*. An asleep replica first enters the *recovering* status before it becomes awake. The recovery period lasts for $\Gamma \geq 2\Delta$ time. In practice, the value of Γ may be adjusted by each replica depending on the amount of data it needs to receive.

We present the recovery protocol for Koala-1 in Algorithm 7. When a replica p_i enters the recovering status at time t, it first computes the current view number v through the global synchronous clock. Then it sends a $\langle \text{RECOVER}, v, t \rangle_i$ message to all replicas, starts a timer with a duration of Γ, and waits for the reply from other replicas. Upon receiving an $\langle \text{RECOVER} \rangle$ message before $t + \Delta$, each replica responds by sending to p_i all its local parameters at $t + \Delta$, along with the VRF evaluation and the proof that were included in the same message when candidate was proposed. Additionally, the replica also sends to p_i all *a-delivered* blocks, the ancestors of candidate, and all messages it receives for view v. When the timer of p_i expires, p_i updates each of its local parameters to the *latest* valid one it has received, e.g., QC with the highest view number. For the pipelined Koala-1 protocol, multiple $prepareQC$ with the highest view number can be formed. In such cases, p_i sets its $prepareQC$ as the one with the highest VRF evaluation. After updating the local parameters, p_i *a-delivers* the block B corresponding to $commitQC$ and all ancestors of B. It then sets its status as awake.

Sketch of correctness. By using the recovery protocol, we can remove the assumptions on stable storage and message delivery. This is mainly because every recovered replica is able to collect all the information needed to ensure the correctness of the protocol. In particular, suppose an honest replica p_i recovers at time t in view v. If a block B is *a-delivered* in a view $v' \leq v$, our recovery protocol ensures that p_i sets its lock as a block no lower than B. In this way, safety is achieved. Namely, if B is *a-delivered* in view $v' < v$, the graded delivery property of VT-GPE (or wT-GPE) ensures that all awake honest replicas set their lock as B at the end of view v'. Thus any replica that recovers after view v' (including p_i) must receive a lock no lower than B from awake replicas. If B is *a-delivered* in view v, p_i will set its lock as B in VT-GPE$_v$ (or wT-GPE$_v$), as p_i receives all messages of view v. Therefore, safety will never be violated. Similarly, if an honest replica is locked on block B in view $v' \leq v$, all awake honest replicas must set their candidate as B at the end of view v' and p_i must collect a candidate such that candidate $= B$ or $view(\text{candidate}) > v'$. Therefore, liveness can be achieved.

Algorithm 7 Recovery protocol for Koala-1

Replica p_i executes the following algorithm.
upon going online at time t **do**
  Let v be the current view.
  broadcast $\langle \text{RECOVER}, v, t \rangle_i$
upon the timer expires **do**
  $prepareQC \leftarrow$ the received $prepareQC$ with the highest view number
  if multiple $prepareQC$ exist for this view **then**
    $prepareQC \leftarrow$ the $prepareQC$ with the highest VRF evaluation
  candidate $\leftarrow$ the block corresponding to $prepareQC$
  $lockedQC \leftarrow$ the received $lockedQC$ with the highest view number
  lock $\leftarrow$ the block corresponding to $lockedQC$
  $commitQC \leftarrow$ the received $commitQC$ with the highest view number
  a-deliver the block corresponding to $commitQC$ and all its ancestors
  set status to *awake* and participate in the protocol's execution
// respond to a recovering replica
upon receiving $\langle \text{RECOVER}, v, t \rangle_j$ before time $t + \Delta$ **do**
  if p_i is awake at $t + \Delta$ **then**
    send to p_j candidate, candidate's ancestors, the VRF evaluation and proof included in the same $\langle \text{INPUT} \rangle$ message when candidate was proposed, $prepareQC$, lock, $lockedQC$, all *a-delivered* blocks, $commitQC$, and all received messages of view v

The communication complexity of the recovery protocol is $O(\kappa n^3 + Ln^2 + lnL)$, where L is the size of a block, κ is the security parameter (i.e., length of the digital signature), and l is the length of the longest chain led by candidate of an awake replica.

Discussion. In our protocols, the leader of a view proposes a new block without including the preceding blocks. We assume that any replica missing the preceding blocks can fetch them from other replicas. In practice, to ensure that all the missing blocks can be fetched, we need to revise the recovery mechanism so that replicas forward *all* received blocks to the recovering replicas. Prior work such as MR [10] uses a similar approach. Such a recovery mechanism may incur high communication costs. It is still an open question whether a more efficient approach can be obtained.

APPENDIX C
PROOF OF IMPOSSIBILITY RESULT

Theorem 1. *In the partially synchronous model, any BFT protocol cannot handle sleepy replicas under the $n \geq 3f + 1$ assumption without the stable storage, where f is the number of byzantine replicas and n is the total number of replicas.*

Proof. Towards a contradiction, we assume that there exists a partially synchronous BFT protocol that tolerates at least one sleepy replica without the stable storage assumption. We construct the strategy of the adversary as follows. First, let one honest replica p_1 sleep and wake up at a chosen time. The other replicas remain awake throughout the execution. Second, divide the awake honest replicas into two groups (each of size f) G_1 and G_2. Third, let there be a large network delay between the two groups until GST.

We use Q_1 to denote G_1, *all* Byzantine replicas, and p_1. We use Q_2 to denote G_2 and *all* Byzantine replicas. It is not difficult to see that $|Q_1| = 2f+1$ and $|Q_2| = 2f$. The quorum size is $2f + 1$.

We now describe an execution E_1 of the protocol. The adversary lets replicas in Q_1 communicate with each other with no network delay (while Q_1 and Q_2 have a large network delay before GST). Without loss of generality, we assume a block B_1 is *a-broadcast* by a replica in Q_1. Since $|Q_1|$ equals the quorum size, all replicas in Q_1 eventually *a-deliver* B_1 at some time t_1. At time t_2 where $t_1 < t_2 < GST$, p_1 falls asleep and immediately wakes up. Since no stable storage is assumed, p_1 does not hold any *proof* that B_1 is *a-delivered*. Meanwhile, replicas of Q_2 begin executing the protocol at time t_2. After time t_2, the adversary lets p_1 communicate with Q_2 with no network delay but p_1 and replicas in Q_1 have a large delay. Now, $|Q_1 \setminus \{p_1\}| = 2f$ and $|Q_2 \cup \{p_1\}| = 2f + 1$. Without loss of generality, we assume a block B_2 is *a-broadcast* by a replica in Q_2. As $|Q_2 \cup \{p_1\}|$ equals the quorum size, p_1 has to first *a-deliver* B_1 before it can *a-deliver* B_2, as otherwise the safety property of the protocol is violated.

There exists another execution E_2. In this execution, replicas in Q_2 begin executing the protocol at time t_2. Again, let p_1 fall asleep and immediately wake up at t_2. After that, the adversary lets p_1 communicate with replicas in Q_2 with no network delay (while $Q_1 \setminus \{p_1\}$ and $Q_2 \cup \{p_1\}$ still have a large network delay before GST). In this case, if B_2 is *a-broadcast*, p_1 may choose to *a-deliver* B_2.

It can be seen that E_1 and E_2 are indistinguishable for p_1. However, p_1 needs to first *a-deliver* B_1 before *a-delivering* B_2 in E_1 for the protocol to be safe while p_1 does not need to *a-deliver* B_1 in E_2. □

We would like to comment that even if we have a powerful "recovery" protocol for p_1 in E_1, the protocol may still be problematic. Indeed, as the number of honest replicas in Q_1 is f, p_1 cannot learn any proof that B_1 is *a-delivered*. For example, the adversary can simply let p_1 communicate with Byzantine replicas and replicas in Q_2. Besides, if up to $2f+1$ honest replicas fall asleep (as assumed in Ebb-and-Flow), it is not clear to us how the quorum size (the number of messages a recovering replica should wait for) should be determined for the recovery protocol when no stable storage is involved.

APPENDIX D
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: The artifact is available in a public Git repository at https://github.com/Spongebob-bear/Koala-NDSS-AE. All the scripts and source codes can also be accessed via the stable URL: https://doi.org/10.5281/zenodo.16956543.

2) Hardware dependencies: The experiments can be run on a single machine that simulates all servers and clients. The recommended hardware specifications are:

- CPU: 4 or more cores
- Memory: 16 GB or more
- Disk: 40 GB of free space (SSD recommended)

3) Software dependencies: The artifact can be tested on Ubuntu Server 22.04 LTS. The following software is required:

- Go1.19.4 linux/amd64 or a higher version
- Python 3.10 or a higher version
- Standard build tools: `git`, `wget`, `zip`, `unzip`, `psmisc`

All dependencies can be installed using the system's package manager (`apt`) or downloaded from the internet. We have provided the related commands in `README.md`.

4) Benchmarks: None.

B. Artifact Installation & Configuration

The installation process involves setting up the required software dependencies, cloning the repository, and building the project.

1. Install the software dependencies listed above using `apt` and `wget` as detailed in the `README.md` file.
2. Git clone the repository:
   ```
   git clone https://github.com/Spongebob-bear/Koala-NDSS-AE.git
   ```
3. Enter the root directory of the project:
   ```
   cd Koala-NDSS-AE
   ```
4. Switch to the stable version:
   ```
   git checkout ndss-ae-2
   ```
5. Build the project:
   ```
   ./scripts/build_offline.sh
   ```

Detailed commands are provided in the `README.md` file. After a successful build, you will find the executables (`server`, `client`, `ecdsagen`) in the root directory of the project.

C. Major Claims

- (C1): HotStuff-mSS achieves higher throughput and lower latency compared to HotStuff that stores all consensus parameters in stable storage, and achieves lower throughput and higher latency compared to HotStuff that stores no parameters in stable storage. This is validated by experiment (E1), which reproduces the results described in Figures 4a-4c, Section 6 of the paper.
- (C2): HotStuff without stable storage is vulnerable to a double-spending attack. This is proven by the experiment (E2), which reproduces the results described in Figure 5, Appendix J of the paper.
- (C3): HotStuff-mSS can defend against the double-spending attack. This is proven by the experiment (E3), which reproduces the results described in Figure 6, Appendix J of the paper.
- (C4): Koala-2 can defend against the double-spending attack. This is proven by the experiment (E4), which reproduces the results described in Figure 7, Appendix J of the paper.

D. Evaluation

We conduct four experiments: HotStuff with different storage options (E1), the double-spending attack on HotStuff without stable storage (E2), the double-spending attack on HotStuff-mSS (E3), and the double-spending attack on Koala-2 (E4). E1, E2, E3, and E4 are used to validate Claim C1, C2, C3, and C4, respectively.

1) Experiment (E1): [HotStuff with different storage options] [10 human-minutes]: We assess latency and throughput for HotStuff under three storage options, i.e., all consensus parameters are stored in stable storage, minimum parameters are stored (i.e., HotStuff-mSS), and no intermediate parameters are stored.

[Notes on scaled-down experiments] The results reported in Figure 4a-4c of our paper require access to the Amazon EC2. In this artifact appendix, we present the procedure of scaled-down experiments using a single machine, which launches four server replicas and one client process.

[How to] Run the following script from the project's root directory.

For HotStuff that stores all parameters in stable storage:

```
./scripts/run_experiment_1.sh 1 50
```

For HotStuff-mSS:

```
./scripts/run_experiment_1.sh 2 30
```

For HotStuff that stores no parameters in stable storage:

```
./scripts/run_experiment_1.sh 3 30
```

The first argument selects the specific sub-experiment. The second argument specifies the duration (in seconds) the script waits for the experiment to complete. The script will first set up the correct configuration, start the 4 server processes, and then launch the client. After starting the script, wait for some time (i.e., the duration specified in the second argument). The script will automatically terminate all processes and calculate the average performance results.

[Results] The scripts will print the throughput and latency of HotStuff at the final line. The following is expected to be shown on the terminal:

```
[Output] Print the performance of the sleepy replica
throughput(tps):4643.004716981132,
↪  latency(ms):698.544117647059
```

Based on our tests on a machine with 4 vCPUs, 16 GB RAM, and 40 GB SSD, the expected performance of HotStuff-mSS relative to the baselines is as follows. Throughput should be approximately 2.4x that of storing all consensus parameters, and 64% that of storing no parameters. Latency should be about 42% that of storing all parameters and 1.7x that of storing no parameters.

On different hardware, the results are expected to vary. Variations in CPU or storage speed (e.g., using a mechanical disk instead of an SSD) are expected to be the major factors that affect the evaluation results. However, the trend is expected to hold, which is sufficient to validate Claim C1.

2) Experiment (E2): [Attack on HotStuff without stable storage] [5 human-minutes]: This experiment aims to reproduce the result shown in Figure 5 of the paper, showing that HotStuff storing no consensus parameters in stable storage is vulnerable to a double-spending attack.

[How to] Run the following script from the project's root directory.

```
./scripts/run_experiment_2_1.sh
```

[Results] The script will first show the setup and client transaction submissions. The crucial part of the output is the log from the sleepy replica (replica 2), which will be printed on the terminal. The following sequence of events is expected to be printed, which confirms a successful double-spending attack.

First, the replica commits the first transaction {From:0,To:1,Value:40} within a block of height 1 before going to sleep.

```
13:39:41 [!!!] Ready to output a value for height 2
...
13:39:41 {"View":0,"Height":1,"TXS":[...,{"ID":100,
"TX": {"From":"0","To":"1","Value":40"},...]}
...
13:39:41 Falling asleep in sequence 5...
```

After waking up, the replica commits a conflicting transaction {From:0,To:2,Value:40} within a block of height 1.

```
13:39:41 sleepTime: 3000 ms
13:39:44 Wake up...
13:39:44 Start the recovery process.
13:39:44 recover to READY
13:39:46 [!!!] Ready to output a value for height 1
...
13:39:46 {"View":0,"Height":1,"TXS":[...,{"ID":100,
"TX":{"From":"0","To":"2","Value":40"},...}]}
```

This sequence demonstrates that the sleepy replica has committed two conflicting blocks at the same height, thus the double-spending attack is performed successfully. The output log should align with Figure 5 in our paper, validating Claim C2.

3) Experiment (E3): [Attack on HotStuff-mSS] [5 human-minutes]: This experiment aims to reproduce the result shown in Figure 6 of the paper, demonstrating that HotStuff-mSS can defend against the double-spending attack.

[How to] Run the following script from the project's root directory.

```
./scripts/run_experiment_2_2.sh
```

[Results] You should observe the following sequence of events:

Before sleep, the sleepy replica commits the first transaction {From:0,To:1,Value:40} within a block of height 1.

```
13:58:35 [!!!] Ready to output a value for height 2
...
13:58:35 {"View":0,"Height":1,"TXS":[...,{"ID":100,
"TX":{"From":"0","To":"1","Value":40"},...}]}
...
13:58:35 Falling asleep in sequence 5...
```

After waking up, the replica knows it was in view 0 and initiates a view change to view 1. The log after the recovery shows that the blocks from view 0, including the one with the first transaction {From:0,To:1,Value:40}, are included in the ledger.

```
13:58:38 Wake up...
13:58:38 Start the recovery process.
13:58:38 recover to the view 1
13:58:38 Starting view change to view 1
...
13:58:45 [!!!] Ready to output a value for height 3
...
13:58:45 {"View":0,"Height":1,"TXS":[...,{"ID":100,
"TX":{"From":"0","To":"1","Value":40},...}]}
```

This sequence demonstrates that the double-spending attack fails. The output log should align with Figure 6 in our paper, validating Claim C3.

4) Experiment (E4): [Attack on Koala-2] [5 human-minutes]: This experiment aims to reproduce the result shown in Figure 7 of the paper, showing that Koala-2 can defend against the double-spending attack.

[How to] Run the following script from the project's root directory.

```
./scripts/run_experiment_2_3.sh
```

[Results] Following sequence of events is expected:

Before sleep, the sleepy replica commits the first transaction {From:0,To:1,Value:40} within a block of height 1.

```
14:17:50 [!!!] Ready to output a value for height 2
...
14:17:50 {"View":0,"Height":1,"TXS":[...,{"ID":100,
"TX":{"From":"0","To":"1","Value":40},...}]}
...
14:17:50 Falling asleep in sequence 5...
```

After waking up, the replica wakes up and executes the recovery protocol of Koala-2. The log shows the messages received by the sleepy replica, such as ECHO1 and ECHO2 messages. The log after recovery shows that the blocks from view 0, including the one with the first transaction {From:0,To:1,Value:40}, are included in the ledger.

```
14:17:54 Wake up...
14:17:54 Start the recovery process.
14:17:54 receive a ECHO1 msg from replica 0
...
14:18:00 receive a TQC msg from replica 1 for view 0
14:18:00 Starting view change to view 1
...
14:18:10 receive a TQC msg from replica 5 for view 1
14:18:10 Starting view change to view 2
...
14:18:10 receive a ECHO2 msg from replica 4
...
14:18:10 recover to READY
...
14:18:10 [!!!] Ready to output a value for height 198
...
14:18:10 {"View":0,"Height":1,"TXS":[...,{"ID":100,
"TX":{"From":"0","To":"1","Value":40},...}]}
```

This sequence demonstrates that the double-spending attack fails. The output log should align with Figure 7 in our paper, validating Claim C4.