TockOwl: Asynchronous Consensus with Fault and Network Adaptability

Minghang Li[1], Qianhong Wu[1], Zhipeng Wang[2], Bo Qin[3,*], Bohang Wei[1], Hang Ruan[1], Shihong Xiong[1], Zhenyang Ding[1]
[1]*Beihang University,* [2]*Imperial College London,* [3]*Renmin University of China*
{liminghang, qianhong.wu}@buaa.edu.cn, zhipeng.wang20@imperial.ac.uk, bo.qin@ruc.edu.cn, {bohang, ruanhang, sy2239215, 18231193}@buaa.edu.cn

Abstract

BFT protocols usually have a waterfall-like degradation in performance in the face of crash faults. Some BFT protocols may not experience sudden performance degradation under crash faults. They achieve this at the expense of increased communication and round complexity in fault-free scenarios. In a nutshell, existing protocols lack the adaptability needed to perform optimally under varying conditions.

We propose TockOwl, the first asynchronous consensus protocol with fault adaptability. TockOwl features quadratic communication and constant round complexity, allowing it to remain efficient in fault-free scenarios. TockOwl also possesses crash robustness, enabling it to maintain stable performance when facing crash faults. These properties collectively ensure the fault adaptability of TockOwl.

Furthermore, we propose TockOwl+ that has network adaptability. TockOwl+ incorporates both fast and slow tracks and employs hedging delays, allowing it to achieve low latency comparable to partially synchronous protocols without waiting for timeouts in asynchronous environments. Compared to the latest dual-track protocols, the slow track of TockOwl+ is simpler, implying shorter latency in fully asynchronous environments.

1 Introduction

Crash fault tolerant (CFT) [70] and Byzantine fault tolerant (BFT) [52] are two essential fault tolerance models in distributed systems. A CFT protocol ensures that the system continues to function normally even when some replicas stop working. Many CFT protocols, such as Paxos [51] and Raft [67], are widely applied in practical systems and services, including distributed databases [9, 20], distributed message queues [5, 49, 76], and container orchestration and scheduling [6]. Once Byzantine faults occur, the security of the CFT protocols will be destroyed.

*Corresponding author

In comparison, a BFT protocol accounts for the existence of Byzantine replicas. The ability to tolerate a certain proportion of Byzantine faults is crucial to ensure the security of numerous protocols, such as federated learning protocols [12,29,54], distributed cryptographic systems [10,25,26,58], and consensus mechanisms [1,15,37,48].

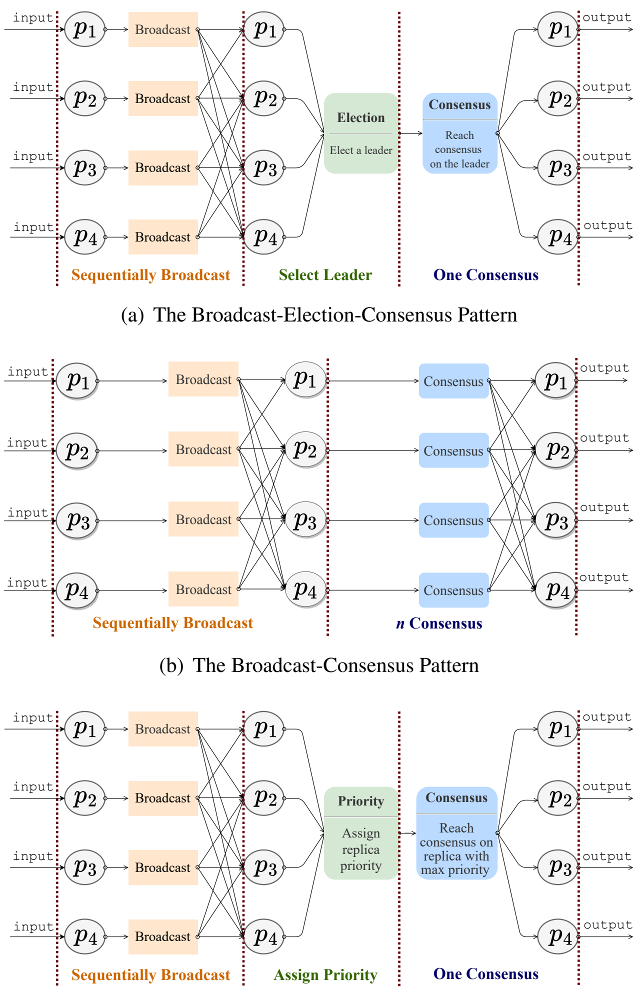
From the perspective of network models, consensus protocols can be categorized into synchronous [2, 3], partially synchronous [18, 80], and asynchronous protocols [27, 63]. The asynchronous model does not make assumptions about the upper bound of message transmission delays, making asynchronous protocols are robust than synchronous and partially synchronous protocols. Therefore, asynchronous BFT protocols are crucial for maintaining security in adversarial environments, such as those involving Byzantine faults and asynchronous networks.

There are many asynchronous BFT protocols [24, 32, 38, 63, 72] that achieve high performance and strong security. Nevertheless, several subtle issues remain. In adversarial environments, do Byzantine faults always exist, or is the network always in an asynchronous state? Furthermore, can the performance of asynchronous BFT protocols be enhanced in more benign environments, such as in the presence of crash faults or within synchronous networks?

1.1 Background and Problem Statement

This section further looks into the above questions.

Crash faults in BFT protocols. Practical consensus systems may experience three states: fault-free, crash fault, and Byzantine fault. Many studies [47, 74] show that crash faults are the most common in practical systems. While CFT protocols cannot handle Byzantine faults, BFT protocols tolerate a certain level of crash faults, maintaining liveness. However, crash faults can still weaken the liveness of many BFT protocols, causing their performance to dramatically degrade. This is not surprising, and we will analyze why this impact occurs later. We test the performance of several known asynchronous BFT



(a) The Broadcast-Election-Consensus Pattern

(b) The Broadcast-Consensus Pattern

(c) Our Pattern: The broadcast-assignment-consensus Pattern

Figure 1: Comparison of asynchronous BFT Patterns.

protocols under fault-free and crashed conditions in experiments (see Section 6). The results show that their performance declines with the occurrence of crash faults. For example, as shown in Figure 5, in sMVBA [38] with 100 replicas on a global network, the latency increases by 100% to 150% when 33 replicas crash, compared to the fault-free scenario.

Asynchronous protocols in synchronous networks. Network conditions are unpredictable due to various changes. Asynchronous protocols guarantee system availability under such conditions, at the cost of more communication rounds than partially synchronous protocols. When the network is synchronous, running an asynchronous protocol leads to performance inefficiencies.

Adaptive asynchronous BFT protocol. An ideal protocol should function effectively in adversarial environments. Existing asynchronous BFT protocols must pay a premium to achieve this, which increases the cost of running the protocol in benign environments. It is difficult to know the specific types of faults and networks, so we expect the protocol to adaptively cope with changes in faults and networks.

Table 1: Comparison of TockOwl and other asynchronous consensus protocols.

Protocol	Model	Complexity		Crash robustness
		Communication	Round	
Speeding-Dumbo [38]	BFT	$O(n^3)$	$O(1)$	✗
FIN [28]	BFT	$O(n^3)$	$O(1)$	✗
CKPS [16]	BFT	$O(n^3)$	$O(1)$	✗
VABA [4]	BFT	$O(n^2)$	$O(1)$	✗
sMVBA [38]	BFT	$O(n^2)$	$O(1)$	✗
DAG-Rider [45]	BFT	$O(n^3)$	$O(1)$	✗
Bullshark [72]	BFT	$O(n^3)$	$O(1)$	✗
Tusk [24]	BFT	$O(n^3)$	$O(1)$	✗
CNV06 [21, 65]	BFT	$O(n^4)$	$O(n)$	✓
WaterBear [82]	BFT	$O(n^3)$	$O(2^n)$	✓
HoneyBadger [63]	BFT	$O(n^3)$	$O(\log n)$	✓
MyTumbler [57]	BFT	$O(n^3)$	$O(\log n)$	✓
QuePaxa [75]	CFT	$O(n^3)$	$O(1)$	✓
TockOwl (this work)	BFT	$O(n^2)$	$O(1)$	✓

1.2 Fault-adaptive Consensus Protocol

Trade-off between complexity and crash robustness. In asynchronous BFT consensus protocols [4, 16, 24, 38, 45, 60, 72] based on multi-value Byzantine agreement (MVBA) and directed acyclic graph (DAG), a broadcast-election-consensus pattern is adopted. As shown in Figure 1(a), each replica independently conducts several rounds of (consistent/reliable) broadcasts for its proposal. Once enough replicas have completed broadcasting, all correct replicas participate in a coin-tossing process to select one replica as the leader and attempt to reach consensus on this leader's proposal. If a replica that has not started working at all is selected as the leader, consensus cannot be achieved in this epoch. From the client's perspective, the underlying consensus network appears to experience jitter, with a sudden increase in transaction latency and a significant decrease in throughput. Consequently, these consensus protocols are vulnerable to crash faults.

We expect that the performance of asynchronous protocols will remain stable in the case that some replicas crash. We refer to this property as *crash robustness* (see Definition 2). Informally, crash robustness means that an increase in the number of crashed replicas does not reduce the success probability of asynchronous consensus.

Asynchronous BFT protocols with a broadcast-consensus pattern can achieve crash robustness at the cost of higher communication and round complexity. As depicted in Figure 1(b), this pattern requires each replica to independently initiate a sub-consensus module after the broadcast phase. For instance, in HoneyBadger [63], each replica is required to initiate an asynchronous binary agreement (ABA), and in MyTumbler [57], each replica must execute a multi-value consensus protocol called SuperMA, which includes a fast track. Since the broadcast and consensus initiated by each replica are independent, crashed replicas cannot prevent the correct replicas from reaching consensus. However, this independence comes at a cost: initiating sub-consensus modules for all replicas requires at least $O(n^3)$ communication complexity, and the parallel execution of n consensus protocols results in $O(\log n)$ round complexity. In contrast, consensus protocols based on the broadcast-election-consensus pattern can achieve $O(n^2)$ communication complexity and constant round complexity, as replicas only need to reach consensus on the leader once. This highlights a trade-off between complexity and crash robustness, leading to the following question:

> *Can we make the BFT protocol fault-adaptive, meaning it maintains high efficiency when there are no faults, maintains stable performance when crash faults occur?*

We design TockOwl, which provides an affirmative answer to this problem. TockOwl is an asynchronous BFT protocol. As shown in Table 1, existing consensus protocols involve trade-offs among fault tolerance, high efficiency, and crash robustness, with none being fault-adaptive. TockOwl achieves fault adaptability by changing the broadcast-election-consensus pattern.

Assign priorities instead of selecting a leader. As illustrated in Figure 1(c), we propose a new pattern called broadcast-assignment-consensus, which uses a common coin to determine replica priorities instead of selecting a leader. We believe that this pattern is generic and helps protocols without crash robustness to achieve this property. In this pattern, each replica is assigned a priority. Correct replicas attempt to reach consensus on the replica with the highest priority in an active set, which consists of replicas that have completed broadcasting. The active set is a subset of all replicas, and non-working replicas are not included in the active set, ensuring that crashed replicas cannot affect the consensus among correct replicas. Note that the highest-priority replica may be mistaken for the leader, but they are inherently different.

- Leader Election: the elected leader may crash, causing the current consensus to fail.
- Priority Assignment: a crashed replica will never be elected. In TockOwl, each replica observes an active replica set AR and attempts to reach consensus on the highest-priority replica within AR. Since a crashed replica is excluded from the active set, it cannot be elected.

On the other hand, the leader is public and undisputed, while the highest-priority replica in the active set is controversial because each replica may observe a different active set. Therefore, reaching consensus on this replica is more challenging than reaching consensus on the leader, especially in asynchronous networks and under the BFT model.

In the CFT model, QuePaxa [75] introduces a method for reaching consensus on the highest-priority replica. This method relies on a critical primitive called threshold synchronous broadcast (tcast) [31]. By continuously invoking tcast, QuePaxa ensures that the AR sets observed by different replicas share a common subset of size of at least $n-f$. If the highest-priority replica is included in this common subset, consensus can be reached. Although implementing tcast in the BFT model using reliable broadcast (RBC) [14, 17] is feasible, it results in high communication complexity when each replica invokes RBC. TockOwl does not require RBC but instead uses a simpler consistent broadcast (CBC) to propagate proposals. After the broadcast phase, TockOwl introduces an exchange phase, where replicas share information without requiring a common subset. The structure of broadcasting followed by exchanging ensures simplicity.

1.3 Network-adaptive Consensus Protocol

Dual-track protocols. Due to the FLP theorem [30], asynchronous consensus requires the introduction of randomized components, which often increases rounds. To enable the protocol to adapt to network changes, dual-track protocols [13, 22, 33, 59] are proposed. These protocols feature both fast and slow tracks: the fast track maintains efficiency in partially synchronous environments, while the slow track ensures liveness in asynchronous environments. There are mainly two types of dual-track protocols. The first type uses timeout delays, activating the slow track only when the fast track fails. The second type uses hedging delays, which allow both tracks to start simultaneously to avoid waiting for a timeout in asynchronous networks.

We primarily focus on protocols using hedging delays. ParBFT [22] is currently the most advanced dual-track protocol, and its two tracks are independent. Since both tracks start simultaneously, they may both output a value. To ensure eventual consistency, an additional consensus instance is required to select between the values from the two tracks, resulting in ParBFT's slow track requiring a total of two asynchronous consensus instances.

TockOwl+: A faster dual-track protocol in asynchronous environments. We design a dual-track protocol called TockOwl+ based on TockOwl. We utilize the highest priority to provide a fast track for TockOwl+, while the slow track is a complete instance of TockOwl. Unlike ParBFT, we integrate the fast track into the slow track, allowing both tracks to share internal state information. This integration makes the protocol more streamlined and effective. Specifically, in TockOwl+, each epoch designates a replica as the leader. The leader has the highest priority, while the priorities of other replicas are determined through a common coin. If the leader works well, the replicas can directly reach consensus on the leader's proposal and output quickly. If not, the replicas can still reach consensus through the complete steps.

TockOwl+ inherits the fault adaptability of TockOwl. Similar to ParBFT [22], TockOwl+ employs hedging delay instead of timeout delay used in BDT [59] and Ditto [33]. The slow track of TockOwl+ consists of an instance of asynchronous consensus, which is more streamlined than the slow tracks of protocols like ParBFT and BDT that require two asynchronous instances.

1.4 Our contributions

In summary, our contributions are as follows:

- We design TockOwl, an asynchronous BFT protocol with fault adaptability. First, TockOwl achieves optimal quadratic communication and constant round complexity, ensuring efficiency in fault-free scenarios. Second, TockOwl possesses crash robustness, ensuring its performance remains unaffected by crash faults.
- We design TockOwl+, an asynchronous BFT protocol that inherits fault adaptability and introduces network adaptability. TockOwl+ is a dual-track protocol that uses hedging delay instead of timeout delay, enabling quick output in partially synchronous environments and requiring no timeout in asynchronous environments. The slow track of TockOwl+ only requires one asynchronous consensus, which is simpler than existing dual-track protocols, meaning TockOwl+ has lower latency in fully asynchronous environments.
- We present two variants of TockOwl. The first variant shortens TockOwl's latency by optimizing the number of broadcast phases at the cost of losing crash robustness. This variant has an expected latency of 10.5 rounds, which is one of the lowest latency asynchronous multi-value BFT protocols. The second variant accepts variable payloads and categorizes proposals as either non-empty or empty based on payload size. This allows the protocol to prioritize committing non-empty proposals, enhancing performance under light or unbalanced load conditions.

2 Related Work

ACS-based asynchronous consensus. Asynchronous common subset (ACS) refers to n participants reaching consensus on a subset of size at least $n-f$. It was first proposed by Ben-Or et al. [10] and applied to the design of asynchronous secure multi-party computation protocols. The ACS protocol proposed by Ben-Or et al. is called BKR. Many protocols [27, 55, 63, 81] have utilized BKR for constructing asynchronous consensus. HoneyBadger [63] uses threshold encryption and Reed-Solomon code to process and broadcast the original transactions, and then uses BKR to reach consensus. Decentruth [78] introduces a variant of BKR called WP-ACS. WP-ACS takes into account the historical weight of replicas, and the values of proposals from replicas with higher weight are more likely to be output. Dumbo [39] and FIN [28] build ACS based on MVBA, that reduces the number of ABA protocols from n to only three as expected.

MVBA-based asynchronous consensus. In MVBA, each replica provides an input that satisfies external validity, and the replicas eventually output one of the values. MVBA is a primitive abstracted by Cachin et al. [16]. They designed an asynchronous atomic broadcast protocol with $O(n^3)$ communication complexity, constant communication rounds, and an optimal 1/3 fault tolerance. VABA [4] is the first MVBA protocol to implement the communication complexity of $O(n^2)$. Speeding-Dumbo [38] further reduces the expected rounds of the protocol while achieving $O(n^2)$ communication complexity. Dumbo [39] constructed ACS based on MVBA for the first time, and this idea is followed by many subsequent works [28, 38]. Furthermore, MVBA can be utilized as a building block to construct more intricate consensus protocols [13, 19, 22, 32].

DAG-based asynchronous consensus. In this type of protocols, each block references $n-f$ blocks that had certificates from the previous round, and blocks are linked in a DAG structure. A notable feature of this type of protocols is that no additional communication is required to reach consensus on top of building the DAG. DAG-Rider [45] is a post-quantum safe asynchronous consensus that is equitable and guarantees that all proposals proposed by the correct replicas will eventually be committed. Bullshark [72] implements a low latency fast track to facilitate rapid output of replicas in good cases. Tusk [24] optimizes DAG-Rider to reduce the latency under normal circumstances.

Protocols for handling hybrid faults. Several protocols [56, 62, 64, 77] involve handling hybrid faults and aim to enhance fault tolerance. XPaxos [56] adopts the cross fault tolerance (XFT) model, achieving an improved level of fault tolerance without increasing resource overhead. The flexible BFT protocol [62] introduces the concept of alive-but-corrupt (a-b-c) faulty replicas that attempt to compromise safety without affecting liveness. This protocol improves overall fault tolerance by reducing Byzantine fault tolerance. The multi-threshold BFT (MT-BFT) protocol [64] separates the safety and liveness threshold definitions and supports both synchronous and asynchronous (or partially synchronous) timing models. The MT-BFT protocol enhances safety fault tolerance by reducing liveness fault tolerance in synchronous networks. The strengthened fault tolerance [77] allows blocks to gradually obtain higher levels of fault tolerance as the blockchain grows. This mechanism introduces stronger safety guarantees in the optimistic period, thus ensuring that blocks can tolerate more than 1/3 faults in non-optimistic periods. These protocols are leader-based and non-asynchronous. If the leader crashes, their performance degrades, and they lose liveness in asynchronous networks.

Crash-robust consensus protocols. Several existing protocols, such as Algorand [35], Goldfish [23], and Mysticeti [8], also introduce priorities or crash-fault-skipping mechanisms into their consensus processes. At a high level, these protocols comprise three main phases: block proposal, block selection, and reaching consensus. The block selection method is critical for ensuring crash robustness.

Similar to TockOwl, Algorand [35] and Goldfish [23] leverage priority assignment to ensure crash robustness. Specifically, Algorand and Goldfish are hybrid protocols that combine Proof-of-Stake (PoS) and BFT. During the block proposal phase, these two protocols use verifiable random functions (VRFs) to determine proposer eligibility. Upon block selection, TockOwl employs a common coin to assign priority, while Algorand and Goldfish derive priority values directly from VRF outputs. After priorities are assigned, Algorand reaches consensus via its Byzantine agreement protocol, BA*, which outputs either a common block or an empty block. Goldfish introduces the GHOST [71] rule to select the branch with the highest weight, ensuring eventual consistency.

Mysticeti [8], a low-latency DAG-based BFT protocol with a communication complexity of $O(n^3)$, does not explicitly adopt a priority assignment mechanism in block selection but introduces a novel *skip pattern* mechanism to exclude crashed replicas. Specifically, the Mysticeti direct decision rule allows replicas to promptly mark a slot as "to-skip" upon observing a skip pattern for that slot, thereby preventing crashed replicas from affecting subsequent block commitments.

However, Algorand, Goldfish, and Mysticeti are not designed to operate in fully asynchronous networks. In terms of efficiency, the latencies of Algorand and Goldfish are constrained by timeout speeds, whereas TockOwl's latency depends solely on network speed.

3 System Model and Definitions

We consider a system consisting of n replicas, denoted as $\{p_1, p_2, \dots, p_n\}$, within which the number of Byzantine replicas is represented by f, satisfying $n = 3f + 1$.

Network assumptions. We assume that each replica is interconnected via a point-to-point authenticated channel, ensuring secure communications. We adopt the asynchronous network model and explicitly do not presuppose any constraints regarding the latency of message transmission.

Cryptographic assumptions. We assume the security of the underlying cryptographic primitives, including hash functions and signatures. Before initiating our series of protocols, it is imperative to establish a threshold cryptography system across all replicas, either through distributed key generation or via a trusted dealer.

Adversary assumptions. We assume that a global and *static* adversary exists, capable of controlling all f Byzantine replicas and having complete control over the network. This implies that the adversary possesses the ability to delay and order messages at will.

Crash robustness. To circumvent the FLP impossibility theorem, asynchronous protocols need to introduce randomness, resulting in a non-zero failure probability.

Definition 1 (Consensus Success Probability)**.** *The consensus success probability of an asynchronous protocol is the probability, $p_k^{(c,b)}$, that the protocol commits a proposal within k communication rounds, where c and b are the proportions of crashed and Byzantine replicas among the total replicas, respectively.*

This probability affects the latency of the protocols. A higher $p_k^{(c,b)}$, implying lower expected rounds, is desirable.

Definition 2 (Crash Robustness)**.** *An asynchronous consensus protocol is of crash robustness if, for any k and $b = 0$, the consensus success probability (i.e., $p_k^{(c,0)}$) does not decrease as c increases, where $0 \leq c < 1/3$.*

State machine replication (SMR). SMR [50, 70] is a technique where each replica maintains a state machine that starts with the same initial state. If the inputs and the order of requests to these state machines are the same, then each state machine will produce the same output.

Definition 3 (SMR)**.** *In a SMR protocol, there exists a set of replicas, and all replicas collectively maintain a linearly growing log and reach consensus on the log's content. The replicas receive and process transactions from clients and interact with each other. Then, each replica outputs a deterministic log, which is a sequence of transactions. A SMR protocol satisfies:*

- ***Safety.*** *If LOG_i and LOG_j are the logs output by any two correct replicas at any time, then either LOG_i is a prefix of LOG_j or LOG_j is a prefix of LOG_i.*
- ***Liveness.*** *Any transaction received by a correct replica will ultimately be recorded in the logs of all correct replicas.*

4 TockOwl: Asynchronous BFT SMR with Fault Adaptability

In this section, we describe TockOwl, which is a SMR protocol that exhibits fault adaptability. Each replica inputs a proposal value v_i in each epoch. The replicas continuously submit proposals and output the values within the proposals to the state machine log.

4.1 Overview

At a high level, as shown in Figure 2, TockOwl consists of three main steps: *three-phase broadcast*, *common coin*, and *Best exchange*.

The first step is the three-phase broadcast. Replicas initiate three consecutive CBCs for their own proposals. Each replica broadcasts its proposal and endeavors to collect $n-f$ votes to form a quorum certificate (QC). Once the QC of the current phase is formed, the replica immediately initiates the next broadcast phase with this QC as the input, similar to the three-phase process in HotStuff [80]. While performing their respective CBCs, replicas collect proposals and QCs from all three phases of other replicas. The proposals are stored in set V, and the QCs from the first/second/third phases are stored in sets $Q1/Q2/Q3$, respectively.

The second step is the common coin. The common coin provides a random value for all replicas, which determines the priority of each replica. Current MVBA protocols select a leader from all replicas based on the common coin, while TockOwl selects the highest-priority replica from a replica set. As shown in Figure 3, we employ a scenario with four replicas to demonstrate the differences in usage. (a) In MVBA protocols, the replica participates in the common coin and elects replica 1 as the leader from all replicas. (b) In TockOwl, the priority order from highest to lowest is replica 1, 3, 2, and 4. The replica confirms that replicas 1, 2, and 4 have completed the three-phase broadcast, forming a set. The replica selects replica 1 as the target replica from the set. (c) The replica confirms that replicas 2, 3, and 4 have completed the three-phase broadcast, forming another set. The replica selects replica 3 as the target replica from the set. Although replica 1 holds the highest priority, it is not in the set.

The third step is the Best exchange. Among the proposals in set V, we can select the replica's proposal with the highest priority, which is referred to as the Best value in V. Similarly, the Best values in $Q1$, $Q2$, and $Q3$ can be obtained. Subsequently, replicas broadcast these Best values and collect Best values from other replicas. When the replicas collect the Best values of $n-f$ replicas, the Best exchange step is completed.

4.2 Terminology and Variables

- **Epoch.** Our protocol operates across epochs. We assume that all message formats are authentic and legitimate, and originate from the current epoch, denoted as e. For a replica in epoch e, if it receives a message from epoch e', it can simply discard the message (if $e' < e$), or it can cache the message until it enters epoch e' (if $e' > e$).
- **Proposal.** At the beginning of each epoch, replicas input a value and package this value into a proposal. Committing a proposal means outputting the value. The proposal value can be a series of transactions or a list of commands from clients. The proposal incorporates a proposal from the previous epoch as its *parent*. This is designed to aid replicas that have not achieved commitment in the previous epoch to successfully commit.

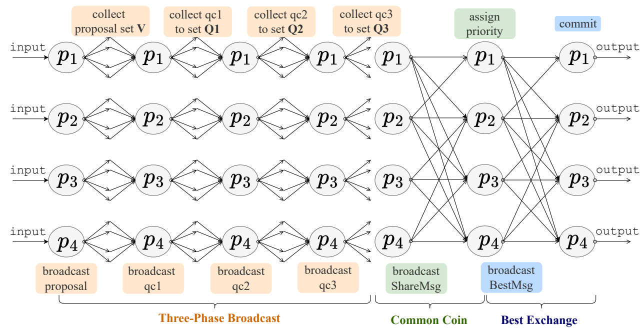
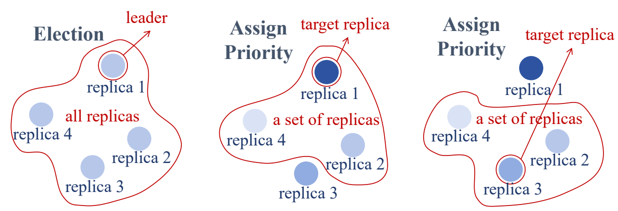


Figure 2: TockOwl structure, which contains three main steps: *three-phase broadcast*, *common coin*, and *Best exchange*.



(a) Elect leader (b) TockOwl case 1 (c) TockOwl case 2

Figure 3: Comparative analysis of methods utilizing the common coin.

- **Quorum Certificate (QC).** Each QC is associated with a proposal. A QC stores a threshold signature, which is used to prove that the proposal has been approved by a majority of replicas. Upon finalizing the three CBCs, replicas can obtain QCs for the three CBCs, denoted as $qc1$, $qc2$, and $qc3$. The subscript of qc indicates its proposer. For example, $qc1_i$ represents the first-phase QC of replica p_i.
- **Priority.** In each epoch, the common coin generates a random *seed*. This *seed* is used to calculate a unique priority for each replica. We assume that these priorities are distinct among replicas, which can be easily achieved in practice. For example, hashing the combination of the *seed* and a replica's sequence number to derive the replica's priority. With this method, the probability of encountering priority collisions is negligible.
- **Priority function.** $\texttt{Pri}_e(proposal/qc)$ represents the priority of the proposer associated with $proposal/qc$, in a given epoch e.
- **Best function.** If R is a set of proposals/QCs, then $\texttt{Best}(R)$ returns the proposal/QC with the highest priority in set R.
- **Variables** $parentQc1$ **and** $parentQc2$**.** Each replica maintains these two variables locally. The variable $parentQc1$

represents the first-phase QC with the highest priority received by the replica in the previous epoch, which is then incorporated into the parent field of the proposal. The variable $parentQc2$ represents the second-phase QC with the highest priority received by the replica in the previous epoch, which is used to verify the validity of the parent field in other proposals.

4.3 Detailed Description

The details of TockOwl are presented in Alg. 1 and Alg. 2.

Three-phase broadcast. Upon entering the current epoch, each replica initiates three consecutive CBCs (Alg. 2, Line 6). Each replica inputs its own proposal into the CBC1 and then inputs the QC output from the CBC1 into the CBC2. This process continues, and every replica eventually obtains three QCs for its proposal. While conducting its own three-phase broadcast, the replica also receives proposals and three phase QCs from other replicas, which are stored in the sets V, $Q1$, $Q2$, and $Q3$, respectively (Alg. 1, Line 15-23).

The replica performs a SAFEPROPOSAL check on the proposals from other replicas (Alg. 1, Line 16). This check mainly involves checking the proposal's parent. The replica compares the priority of the parent in the proposal with the priority of the locally maintained $parentQc2$, and this check passes only when the former is not less than the latter. If the SAFEPROPOSAL check fails, the replica refuses to vote. When each of the four sets contains at least $n-f$ elements, the replica outputs `Finish` from the three-phase broadcast.

Common coin. The replica initiates the common coin by broadcasting a share of the coin. In this process, we set the coin threshold to $n-f$ and use the BV-broadcast proposed in Mostefaoui's ABA [66]. Once the replica receives $n-2f$ shares and has not yet broadcast its own share, it then broadcasts its share. Eventually, each replica receives $n-f$ shares, allowing for the derivation of a common and random value, denoted as *seed*. Based on this *seed*, the replica calculates the priorities for all replicas utilizing a transparent and public-known method. Given that the *seed* generated by every replica is identical, their perceptions of the calculated priorities are also consistent. Moreover, we assume that the priorities are unique. After obtaining the *seed*, the replica terminates the three-phase broadcast and stops participating in the CBC that other replicas have not completed (Alg. 2 Line 14).

Best exchange. After the priorities are determined, the replica utilizes the `Best` function to identify and select the proposal or QC with the highest priority among the sets V, $Q1$, $Q2$, and $Q3$. Following this selection, the replica broadcasts these prioritized values through a *BestMsg* message (Alg. 2, Line 15). Concurrently, the replica receives *BestMsg* messages from other replicas, incorporating their proposals and three phase QCs into corresponding sets V, $Q1$, $Q2$, and $Q3$ (Alg. 2 Line 16).

Algorithm 1 Three-phase broadcast (for epoch e, replica p_i)

Global: $V, Q1, Q2, Q3$
Input: $proposal_i$

// CBC Process
1: **upon** receiving $(id,*,*)$ from Main Broadcast Process **do**
2: broadcast $(id,*,*)$
3: **upon** receiving $(id,*,*)$ from p_j **do**
4: calculate the threshold signature and vote for p_j
5: **upon** receiving $n-f$ vote for $(id,*,*)$ **do**
6: aggregate signatures to get $newQc$
7: output $(id,*,newQc)$ to Main Broadcast Process

// Main Broadcast Process
8: input (`cbc1`, $proposal_i, null$) to $CBC1_i$
9: **upon** outputting (`cbc1`, $proposal_i, qc1_i$) from $CBC1_i$ **do**
10: input (`cbc2`, $H(proposal_i), qc1_i$) to $CBC2_i$
11: **upon** outputting (`cbc2`, $H(proposal_i), qc2_i$) from $CBC2_i$ **do**
12: input (`cbc3`, $H(proposal_i), qc2_i$) to $CBC3_i$
13: **upon** outputting (`cbc3`, $H(proposal_i), qc3_i$) from $CBC3_i$ **do**
14: broadcast (`last`, $H(proposal_i), qc3_i$)

15: **upon** receiving (`cbc1`, $proposal_j, null$) from p_j **do**
16: **if** SAFEPROPOSAL($proposal_j$) **then** // every replica verifies whether this rule holds before participating in $CBC1_j$
17: add $proposal_j$ to V
18: **upon** receiving (`cbc2`, $H(proposal_j), qc1_j$) from p_j **do**
19: add $qc1_j$ to $Q1$
20: **upon** receiving (`cbc3`, $H(proposal_j), qc2_j$) from p_j **do**
21: add $qc2_j$ to $Q2$
22: **upon** receiving (`last`, $H(proposal_j), qc3_j$) from p_j **do**
23: add $qc3_j$ to $Q3$
24: **upon** $|V|, |Q1|, |Q2|, |Q3|$ are all not less than $n-f$ **do**
25: **Output** `Finish`

To prevent the redundant broadcasting of proposals, the *BestMsg* message transmits only the hash value of the proposal, rather than the proposal itself. Upon receiving a *BestMsg* message, the replica retrieves the original proposal through the GETPROPOSALBYHASH process (Alg. 2, Line 17). If the replica does not have the original proposal locally, it requests the proposal from the sender of the *BestMsg*. The logic of GETPROPOSALBYHASH is independent of the consensus logic, and many consensus protocols use similar modules for block and transaction synchronization. Byzantine replicas may send a *BestMsg* message containing a meaningless hash value, thereby preventing replicas from retrieving the corresponding proposal. In such cases, a *BestMsg* message for which the original proposal cannot be obtained is deemed invalid.

Upon receiving at least $n-f$ *BestMsg* messages, the replica uses the UPDATEPARENT function to assign the values of `Best`$(Q1)$ and `Best`$(Q2)$ to the variables $parentQc1$

Algorithm 2 TockOwl protocol (for epoch e, replica p_i)

Initialization: If $e = 1$, then $parentQc1, parentQc2 \leftarrow null$. $V, Q1, Q2, Q3 \leftarrow \{\}$. Let *value* represent the value input by p_i.

// Utilities

1: **procedure** UPDATEPARENT($qc1$,$qc2$)

2: $parentQc1 \leftarrow qc1$, $parentQc2 \leftarrow qc2$

3: **procedure** SAFEPROPOSAL($proposal(e, value, qc)$)

4: **return** $\texttt{Pri}_{e-1}(qc) \geq \texttt{Pri}_{e-1}(parentQc2)$ // the priority of *null* defaults to 0

// Three-phase broadcast

5: $proposal \leftarrow (e, value, parentQc1)$

6: input *proposal* to Three-phase broadcast

7: **upon** outputting `Finish` from Three-phase broadcast **do**

8: broadcast $ShareMsg(coinShare)$ **if** not

// Common Coin

9: **upon** receiving *ShareMsg* messages from $n-2f$ replicas **do**

10: broadcast $ShareMsg(coinShare)$ **if** not

11: **wait** until receiving *ShareMsg* messages from $n-f$ replicas

12: $seed \leftarrow CommonCoin(e)$

13: determine the priority for each replica based on *seed*

14: stop participating in unfinished CBC

15: broadcast $BestMsg(H(\texttt{Best}(V)), \texttt{Best}(Q1), \texttt{Best}(Q2), \texttt{Best}(Q3))$

// Best exchange

16: **upon** receiving $BestMsg(h, bqc1, bqc2, bqc3)$ from p_j **do**

17: $bv \leftarrow$ GETPROPOSALBYHASH(h)

18: add $bv, bqc1, bqc2, bqc3$ to $V, Q1, Q2, Q3$, respectively

19: **wait** until receiving *BestMsg* messages from $n-f$ replicas

20: UPDATEPARENT($\texttt{Best}(Q1)$, $\texttt{Best}(Q2)$)

21: **if** $\texttt{Pri}_e(\texttt{Best}(V)) = \texttt{Pri}_e(\texttt{Best}(Q3))$ **then**

22: **Commit** the proposal in $\texttt{Best}(V)$ and its uncommitted ancestor proposals

and $parentQc2$ (Alg. 2, Line 20). These variables are used for referencing and verifying proposals for the forthcoming epoch. UPDATEPARENT provides a secure method for updating, and in conjunction with SAFEPROPOSAL, it ensures cross-epoch safety. Although faulty replicas might not update their $parentQc1$ and $parentQc2$ according to the UPDATEPARENT method, SAFEPROPOSAL ensures that proposals with incorrect parents do not gain the approval of correct replicas.

The replica determines whether the commit condition, $\texttt{Pri}_e(\texttt{Best}(V)) = \texttt{Pri}_e(\texttt{Best}(Q3))$, is satisfied or not. If this condition holds, it indicates that $\texttt{Best}(V)$, $\texttt{Best}(Q1)$, $\texttt{Best}(Q2)$, and $\texttt{Best}(Q3)$ originate from the same replica, and this replica's proposal is committed. Moreover, once a replica identifies the QC of the highest-priority replica among all replicas within its own Q3 set, it can immediately commit a proposal, serving as a shortcut to the protocol's output.

Intuition of three-phase broadcast. TockOwl requires a three-phase broadcast instead of a two-phase broadcast because a two-phase broadcast cannot ensure cross-epoch safety.

Consider an example where the protocol uses a two-phase broadcast and the commit condition is $\texttt{Pri}_e(\texttt{Best}(V)) = \texttt{Pri}_e(\texttt{Best}(Q2))$. If a correct replica p_i meets the commit condition in epoch e and commits $proposal_l$, then its $\texttt{Best}(Q1) = qc1_l$ and $\texttt{Best}(Q2) = qc2_l$. For another correct replica p_j, $\texttt{Best}(V) = proposal_l$, $\texttt{Best}(Q1) = qc1_l$, and $\texttt{Best}(Q2) = qc2_m$ (where $qc2_m$ has a lower priority than $qc2_l$). In epoch $e+1$, both p_i and p_j will reference $qc1_l$ as the parent of their proposals, while a faulty replica p_k references $qc1_m$. Although p_i can verify that the proposal of p_k is illegal (since p_i has $qc2_l$), p_j cannot. If p_j and other replicas reach consensus on p_k's proposal in epoch $e+1$ and commit it, inconsistency will occur.

We provide a protocol called TockCat that uses a two-phase broadcast and has quadratic communication complexity in Appendix B, at the cost of lacking crash robustness. On the other hand, if we require both crash robustness and two-phase broadcast, we can derive a variant of the TockOwl with cubic communication complexity. In other words, there is a trade-off between a two-phase broadcast, quadratic communication, and crash robustness. It is not yet clear whether there exists a protocol that simultaneously satisfies all three properties.

4.4 Two variants of TockOwl

TockCat: Asynchronous BFT SMR with low latency. We present TockCat, a variant of TockOwl, which utilizes a common coin to select a leader and employs an additional broadcast phase to exchange the leader's value. TockCat achieves quadratic communication and an expected latency of only 10.5 rounds, making it as fast as the current lowest-latency asynchronous multi-valued Byzantine consensus protocol. The specific details of TockCat are provided in Appendix B.

TockWhale: Asynchronous BFT SMR with a preference for non-empty proposals. We introduce TockWhale, a variant of TockOwl, designed to improve the quality of committed proposals by prioritizing non-empty ones. TockWhale assigns the lowest priority to empty proposals, allowing the protocol to preferentially commit non-empty proposals. The lowest priority grants the replica the right to forgo its proposal from being committed in the current epoch. While forgoing the chance to be committed might seem counterintuitive, it enhances efficiency in systems with unbalanced or light loads, such as blockchain-based electronic contract depositories [40, 61] and intellectual property protection systems [42, 44]. In these systems, replicas must receive client transactions to form valid proposals. However, if no transactions are received within a given time, replicas are forced to propose empty values to maintain protocol liveness. These empty proposals, although necessary, are otherwise meaningless and can compete with non-empty proposals, reducing the likelihood of the latter being committed. A detailed discussion of TockWhale can be found in the full version of the paper [53].

4.5 Protocol Correctness

If $Proposal1$ is an ancestor proposal of $Proposal2$, we say that $Proposal2$ extends $Proposal1$.

Lemma 1. *In epoch e, for any correct replicas p_i, p_j, p_k, p_l that have completed the Best exchange,* $\texttt{Pri}_e(\texttt{Best}(V_i)) \geq \texttt{Pri}_e(\texttt{Best}(Q1_j)) \geq \texttt{Pri}_e(\texttt{Best}(Q2_k)) \geq \texttt{Pri}_e(\texttt{Best}(Q3_l))$ *holds.*

Proof. For any replica p_l, let its $\texttt{Best}(Q3_l)$ be $qc3_m$. $qc3_m$ is a third-phase QC, which means at least $n-2f$ correct replicas voted for $qc2_m$ and added it to their $Q2$ sets. These correct replicas broadcast messages $BestMsg(*,*,bqc2,*)$ during the Best exchange step, where $\texttt{Pri}_e(bqc2) = \texttt{Pri}_e(\texttt{Best}(Q2)) \geq \texttt{Pri}_e(qc2_m)$. Due to quorum intersection, any correct replica p_k will receive at least one $BestMsg(*,*,bqc2,*)$ message and add $bqc2$ to its $Q2$ set. Thus, $\texttt{Pri}_e(\texttt{Best}(Q2_k)) \geq \texttt{Pri}_e(bqc2) \geq \texttt{Pri}_e(qc2_m) = \texttt{Pri}_e(\texttt{Best}(Q3_l))$ holds.

Similarly, $\texttt{Pri}_e(\texttt{Best}(V_i)) \geq \texttt{Pri}_e(\texttt{Best}(Q1_j))$ and $\texttt{Pri}_e(\texttt{Best}(Q1_j)) \geq \texttt{Pri}_e(\texttt{Best}(Q2_k))$ can be obtained. □

Lemma 2. *In epoch e, after a replica reaches the commit condition, then* $\texttt{Pri}_e(\texttt{Best}(Q1_i)) = \texttt{Pri}_e(\texttt{Best}(Q2_i)) = \texttt{Pri}_e(\texttt{Best}(Q1_j)) = \texttt{Pri}_e(\texttt{Best}(Q2_j))$ *holds for any correct replicas p_i, p_j.*

Proof. This follows directly from Lemma 1 and the commit condition (Alg. 2 Line 21). □

Lemma 3. *If correct replicas p_i and p_j commit $Proposal1$ and $Proposal2$ with epoch number e in epoch e, respectively, then $Proposal1 = Proposal2$.*

Proof. According to Lemma 2, after both p_i and p_j commit a proposal, $\texttt{Pri}_e(\texttt{Best}(Q1_i)) = \texttt{Pri}_e(\texttt{Best}(Q1_j)) = \texttt{Pri}_e(\texttt{Best}(V_i)) = \texttt{Pri}_e(\texttt{Best}(V_j))$ holds. Therefore, the proposals they commit are consistent. □

Theorem 1 (Safety)**.** *In TockOwl, if a correct replica p_i commits $Proposal1$ with epoch number e in epoch e, and a correct replica p_j commits $Proposal2$ with epoch number e' in epoch e', then either $Proposal1$ extends $Proposal2$ or $Proposal2$ extends $Proposal1$.*

Proof. When $e = e'$, the proof can be directly derived from Lemma 3. Without loss of generality, we assume $e < e'$. In epoch e, p_i commits $Proposal1$. Let p_k be the proposer of $Proposal1$, and let the corresponding three-phase QCs be $qc1_k$, $qc2_k$, and $qc3_k$. According to Lemma 2 and the UPDATEPARENT rule, all correct replicas set their own $parentQc1 = qc1_k$, and $parentQc2 = qc2_k$. Let the priority of p_k as pri.

Next, we prove that there does not exist a first-phase QC with a priority greater than pri in epoch e. Using proof by contradiction, we assume that such a QC exists in replica p_l's set $Q1_l$, then $\texttt{Pri}_e(Best(Q1_l)) > pri = \texttt{Pri}_e(Best(Q1_i))$, which contradicts Lemma 2.

In epoch $e+1$, if a faulty replica references a proposal with a priority lower than pri, then $\texttt{Pri}_e(proposal.parentQc) < pri = \texttt{Pri}_e(qc2_k) = \texttt{Pri}_e(parentQc2)$ holds. This means that this proposal will not pass the SAFEPROPOSAL check in Alg. 2, making the proposal invalid.

In other words, in epoch $e+1$, all valid proposals will reference $Proposal1$, so all subsequently committed proposals will extend from $Proposal1$. □

Lemma 4. *If all correct replicas input a value in epoch e, then each correct replica eventually receives at least $n-f$ $BestMsg$ messages and subsequently enters epoch $e+1$.*

Proof. If all correct replicas broadcast $ShareMsg$ messages, then each correct replica obtains the common coin value and subsequently broadcasts a $BestMsg$ message. Eventually, all correct replicas can receive enough $BestMsg$ messages. Therefore, the key to this lemma is to prove that all correct replicas will broadcast $ShareMsg$ messages. We consider three cases:

- **Case1:** All correct replicas obtain outputs from the three-phase broadcast. In this case, these correct replicas will broadcast $ShareMsg$ messages.
- **Case2:** Some correct replicas obtain output from the three-phase broadcast, while other correct replicas do not. Assume that time t is the moment when the first correct replica receives $n-f$ valid $ShareMsg$ messages. Before time t, at least $n-2f$ correct replicas have already broadcast $ShareMsg$ messages. Subsequently, the correct replicas that have not yet broadcast will receive at least $n-2f$ $ShareMsg$ messages and broadcast $ShareMsg$ messages.
- **Case3:** No correct replica obtains output from the three-phase broadcast. This is impossible because faulty replicas cannot provide $n-2f$ valid $ShareMsg$ messages.

Thus, in any case, all correct replicas will broadcast $ShareMsg$ messages, and there must exist some correct replicas that obtain outputs from the three-phase broadcast. □

Theorem 2 (Liveness)**.** *In epoch e, the probability that a correct replica commits a proposal is greater than 2/3.*

Proof. Let the replica with the highest priority among the n replicas be p_l. According to Lemma 4, some correct replicas definitely obtain outputs from the three-phase broadcast. For any such replica p_i, $|Q3_i| \geq n-f$, if the third-phase QC of p_l appears in $Q3_i$, then p_i can definitely commit the proposal of p_l in the current epoch. Therefore, the probability is $(n-f)/n > 2/3$. □

Lemma 5 (Crash Robustness)**.** *Let k represent the number of communication rounds, and c and b represent the proportions* *of crashed and Byzantine replicas among the total replicas, respectively. For any k and $b=0$, the consensus success probability (i.e., $p_k^{(c,0)}$) does not decrease as c increases, where $0 \le c < 1/3$.*

Proof. TockOwl uses the priority assignment. With priority assignment, a crashed replica will never be elected. Due to $b = 0$, the elected replica is not a Byzantine one, so it must be a correct one. Since $0 \le c < 1/3$, the proposal of this elected replica will be successfully committed in one epoch (9 rounds). Therefore, when $k < 9$, $p_k^{(c,0)} = 0$ and when $k \ge 9$, $p_k^{(c,0)} = 1$. Hence, $p_k^{(c,0)}$ remains constant as c increases, provided that $0 \le c < 1/3$. This implies TockOwl has crash robustness. □

Lemma 6 (Efficiency)**.** *TockOwl exhibits quadratic communication complexity and an expected constant round latency.*

Proof. The protocol uses all-to-all broadcasting in each round, so it has $O(n^2)$ communication complexity. The protocol requires 9 rounds per epoch, and the probability of committing in each epoch is 2/3. This leads to an expected latency of $9 \times 3/2 = 13.5$ rounds. □

Algorithm 3 TockOwl+ (for epoch e, replica p_i, leader p_l)

If $e = 1$, then $parentQc1, parentQc2 \leftarrow null$. Let $value$ represent the value input by p_i.

$bd = false$ // bd indicates whether p_i has broadcast data
$proposal \leftarrow (e, value, parentQc1)$

// Fast Track
if p_i is leader **then**
  input $proposal_l$ to Three-phase broadcast
  upon obtaining $(qc1_l, qc2_l, qc3_l)$ from Three-phase broadcast **do**
    broadcast $HaltMsg(H(proposal_l), qc1_l, qc2_l, qc3_l)$
upon receiving $HaltMsg(hash, qc1_l, qc2_l, qc3_l)$ from p_j **do**
  UpdateParent($qc1_l, qc2_l$) // see Alg. 2, Line 1
  Commit $proposal_l$ and its uncommitted ancestor proposals
  if bq **then**
    broadcast $HaltMsg(hash, qc1_l, qc2_l, qc3_l)$ **if** not
  enter epoch $e + 1$

// Slow Track
wait until receiving the timer T expires
  $bd = true$
  activate TockOwl$_e$($proposal.value$) // SlowTrack is an instance of TockOwl (see Alg. 2)
  enter epoch $e + 1$

5 TockOwl+: Asynchronous BFT SMR with Network Adaptability

5.1 Overview

Building on TockOwl, we introduce TockOwl+, a protocol that incorporates a fast track. In TockOwl+, a replica is preselected as the leader and assigned the highest priority. The leader can be chosen using various methods employed by partially synchronous protocols, such as fixed leader [18, 43, 73], rotating leader [36, 80], or periodic rotation based on

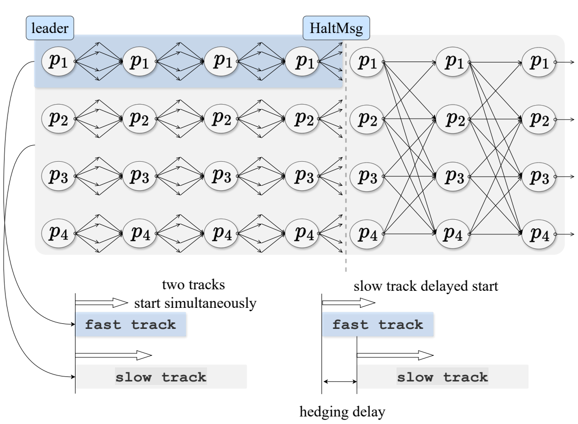


Figure 4: The structure of TockOwl+, which is a dual-track protocol that includes both a fast and a slow track.

performance history [7]. In TockOwl+, the leader is selected using the Round-Robin [36, 80] method.

If the leader completes the three-phase broadcast step, all replicas will observe the leader's proposal and first/second phase QC during the Best exchange step. Therefore, once a replica detects the leader's third-phase QC in its $Q3$ set, it can immediately commit the leader's proposal without waiting for the Best exchange step to conclude.

TockOwl+ is a dual-track protocol [13, 22, 54, 59], characterized by the integration of both a fast and a slow track. The fast track enhances efficiency within partially synchronous environments, while the slow track guarantees liveness in asynchronous environments. Generally, the fast track drives the protocol's progress. However, when network asynchrony disrupts the fast track, the slow track survives with the protocol's advancement. As illustrated in Figure 4, TockOwl constitutes the slow track of TockOwl+, ensuring its liveness.

5.2 Detailed Description

The complete description of TockOwl+ is shown in Alg. 3. TockOwl+ operates in epochs. We assume that all message formats are authentic and legitimate, and originate from the current epoch, denoted as e.

Fast track. At the beginning of each epoch, the fast track is initiated first. The leader attempts to propagate its proposal using a three-phase broadcast. Optimistically, the leader obtains a third-phase QC for its proposal and then broad-

Table 2: Comparison of the TockOwl+ protocol and recent dual-track protocols.

Protocol	Communication complexity		The structure of slow track	Track switching delay
	Fast track	Slow track		
PABC [68]	$O(n)$	$O(n^3)$	Two asynchronous consensus instances	timeout delay
Bolt-Dumbo [59]	$O(n)$	$O(n^2)$	Two asynchronous consensus instances[(2)]	timeout delay
Ditto [33]	$O(n)$	$O(n^2)$	One asynchronous consensus instance	timeout delay
Abraxas [13]	$O(n^2)$	$O(n^2)$	One asynchronous consensus instance	timeout delay[(4)]
ParBFT1 [22] (pattern 1)[(1)]	$O(n^2)$	$O(n^2)$	Two asynchronous consensus instances [(3)]	none
ParBFT2 [22] (pattern 2)	$O(n)$	$O(n^2)$	Two asynchronous consensus instances	hedging delay
TockOwl+ (pattern 1)	$O(n^2)$	$O(n^2)$	One asynchronous consensus	none
TockOwl+ (pattern 2)	$O(n)$	$O(n^2)$	One asynchronous consensus	hedging delay

[(1)]Pattern 1 means that both tracks start at the same time, and pattern 2 means that the slow track starts with delay. [(2)]One asynchronous consensus instance is used for pace synchronization, and another consensus instance is used to reach consensus on transactions. [(3)]One asynchronous consensus instance is used to reach consensus on transactions, and another consensus instance is used to determine the value of whether to use the fast track or the slow track. [(4)]In Abraxas, the fast track and the slow track operate simultaneously. However, during the fast track's operation, the slow track processes transactions without committing them. The slow track only commits transactions when it detects that the fast track makes no progress, so we consider that Abraxas still employs a timeout delay.

casts a *HaltMsg* message. Replicas that receive the *HaltMsg* message can immediately commit the leader's proposal. If $bq = true$, the replica also broadcasts the *HaltMsg* message to assist uncommitted replicas. The variable *bq* identifies whether the slow track has been activated. To avoid introducing quadratic complexity into the fast track, replicas only broadcast *HaltMsg* messages after the slow track has been activated. Furthermore, for a replica p_i that has already committed in the fast track, if it receives a message from p_j regarding the slow track, p_i must forward the *HaltMsg* message to p_j. A number of asynchronous [22, 38] and partially synchronous protocols [80] adopt similar approaches to help replicas that have not committed or lack critical blocks to commit.

The replica updates *parentQc*1 and *parentQc*2 to the leader's first-phase and second-phase QCs through the UPDATEPARENT function. The UPDATEPARENT and SAFEPROPOSAL rules collectively ensure the cross-epoch safety of TockOwl+. These rules maintain consistency regardless of whether they are applied in the fast track or the slow track. Specifically, the UPDATEPARENT rule guarantees that correct replicas can accurately update their proposed parent, while the SAFEPROPOSAL rule ensures that proposals referencing an incorrect parent are not approved.

Slow track. The slow track is a complete single-epoch instance of TockOwl. Each replica locally maintains a timer, T. Once T times out, replicas set $bq = true$ and initiate the slow track. From that point onward, the fast track and slow track proceed in parallel. Note that in TockOwl, when a replica receives $n - f$ *ShareMsg* messages, it stops voting in any unfinished CBCs (Alg. 2 Line 14), including the CBC initiated by the leader in the fast track. Moreover, the leader has the highest priority, and the `Best` function returns the leader's value unless the set does not contain the leader's value.

Safety overview. If we consider only the fast track or the slow track, then the safety of TockOwl+ is easily obtained. We mainly need to prove that the protocol remains safe even if one replica commits a proposal from the fast track while another replica commits a proposal from the slow track. We defer the complete proof to Appendix A.

5.3 Protocol Analysis

Hedging delays. As shown in Table 2, TockOwl+ offers certain advantages compared to state-of-the-art dual-track protocols. In TockOwl+, the setting of timer T is flexible. Typically, we set the time of T to be a multiple of the maximum network delay. This ensures that under optimistic conditions, the protocol outputs through the fast track with $O(n)$ communication complexity, thereby enhancing performance. Alternatively, we can even set the time to 0, meaning that the fast track and slow track start simultaneously to avoid timeouts. QuePaxa [75] refers to this as a hedging delay, which differs from the timeout delay in Bolt-Dumbo [59] and Ditto [33]. When a timeout is triggered, the protocol stops the fast track, while triggering a hedging does not. Incorrectly estimating the timeout delay, causing the slow track to start early, leads to increased communication and latency. In contrast, incorrectly estimating the hedging delay only leads to increased communication.

Faster slow track. The slow track of TockOwl+ is more efficient, as it is constructed from a single asynchronous consensus, whereas the slow tracks in Bolt-Dumbo [59] and ParBFT [22] require two asynchronous consensus instances. Consequently, TockOwl+ exhibits lower latency in asynchronous environments.

6 Implementation and Evaluation

Implementation details. We implement and test TockOwl, TockOwl+, and TockCat in a Wide Area Network (WAN) environment. In the same project, we also implement BKR [11], sMVBA [38], and ParBFT [22]. All protocols are implemented in Golang, with the project is forked from the open-source implementation[1] of Dory [83]. To ensure that performance differences stem from the logical differences in the consensus mechanisms themselves, the implementations utilize unified underlying components. Specifically, LevelDB[2] is used for storing transactions and blocks, and Boldyreva's pairing-based threshold scheme on the BN256 curve, implemented in Kyber[3], is employed for threshold signatures and coin tossing. Replica communications are facilitated via gRPC[4]. The BKR protocol consists of reliable broadcast (RBC) and asynchronous binary agreement (ABA), and we use the RBC version from [41] and the ABA version from [66]. The slow track of ParBFT includes multi-valued Byzantine agreement (MVBA) and ABA, with MVBA sourced from [38] and ABA from [66].

We deploy a consensus network on Amazon Web Services, using m7g.8xlarge instances across 5 different AWS regions: N. Virginia (us-east-1), Sydney (ap-southeast-2), Tokyo (ap-northeast-1), Stockholm (eu-north-1), and Frankfurt (eu-central-1). They provide 15Gbps of bandwidth, 32 virtual CPUs on AWS Graviton3 processor, and 128GB memory and run Linux Ubuntu server 22.04.

We implement the mempool of Dumbo-NG [32] to facilitate the synchronization of data blocks among replicas. Dumbo-NG decouples the process of broadcasting and consensus, a strategy that has been shown to improve throughput in various protocols [19, 24, 46, 72, 79]. Specifically, Dumbo-NG has a broadcast module that propagates transactions among replicas, and we use this module for transaction synchronization. Dumbo-NG also has a consensus module that orders batches of transactions, and we instantiate this module with our asynchronous protocols. ParBFT similarly adopts this strategy in its experiments. Each transaction in the mempool is set to a size of 250 bytes. When missing blocks are detected, a replica sends a message $CallHelp$ to other replicas by the block synchronizer. Correct replicas run a $Helper$ process to respond to $CallHelp$.

Throughput is calculated as the average number of transactions committed per second. Latency is calculated as the total time from when a transaction is proposed to when it is committed, including the time for transaction consensus as well as the time spent synchronizing and waiting in the mempool. For each data point, we conduct three experiments, each lasting five minutes, and report the average throughput and latency. The error bars represent one standard deviation.

[1] https://github.com/xygdys/Dory-BFT-Consensus
[2] https://github.com/syndtr/goleveldb
[3] https://github.com/dedis/kyber
[4] https://github.com/grpc/grpc-go

Experimental composition. We conduct tests in small-scale (10 replicas) and large-scale (100 replicas) networks. The experiments are mainly divided into two parts:

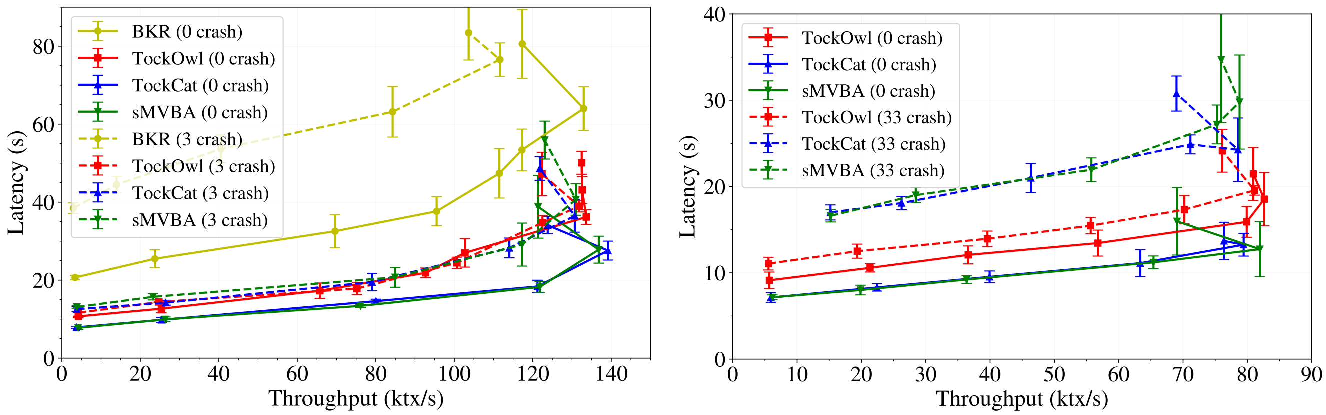
- We compare our protocols, TockOwl and TockCat, with the well-known BKR and sMVBA protocols in terms of fault adaptivity. BKR demonstrates crash robustness, while sMVBA exhibits quadratic communication complexity. Both crash robustness and quadratic communication complexity are fundamental characteristics of TockOwl. Moreover, TockCat shares similar properties with sMVBA but requires fewer rounds. As such, TockCat is also included in this test.
- We compare our TockOwl+ protocol with ParBFT, the state-of-the-art dual-track protocol, in terms of network adaptivity. We do not compare TockOwl+ with protocols that have timeout delay, such as BDT and Ditto, because they are not in the same category as TockOwl+. These protocols require the setting of timeout parameters, and the accuracy of the timeout parameter settings affects the performance of the protocols. In practice, it is difficult to accurately set the timeout parameters, making an experimental comparison between BDT, Ditto, and TockOwl+ less practically meaningful.

6.1 Tests on Fault Adaptability

We test TockOwl, TockCat, BKR, and sMVBA in three environments: no fault, crash faults, and Byzantine faults. The crash and Byzantine faults only target the consensus process. In other words, faulty replicas exhibit faulty behavior in the consensus but still participate normally in the operation of the mempool. Mempool faults that have no effect on consensus can be ignored, and other mempool faults can be simulated by consensus action failure. Crash-free mempool simplifies the engineering implementation without affecting the experimental result and analysis. Moreover, we do not provide performance data for BKR with 100 replicas, as BKR's latency exceeds 200 seconds even in fault-free conditions. Under such circumstances, it no longer makes sense to compare BKR's performance with other protocols.

No fault and crash faults. We first change the number of faulty replicas and evaluate the performance of the protocols under fault-free and replica crash conditions. Both TockCat and sMVBA have output shortcuts, allowing them to output in 6 rounds when there are no faulty replicas. To show the performance of the protocols under normal conditions, we enable the output shortcut for both TockCat and sMVBA (see Figure 5).

When there are no faulty replicas, the performance of TockCat and sMVBA is comparable, and both outperform TockOwl. BKR performs the worst, which is not surprising given



(a) 10 Replicas.

(b) 100 Replicas.

Figure 5: Performance of TockOwl under no fault and crash faults.

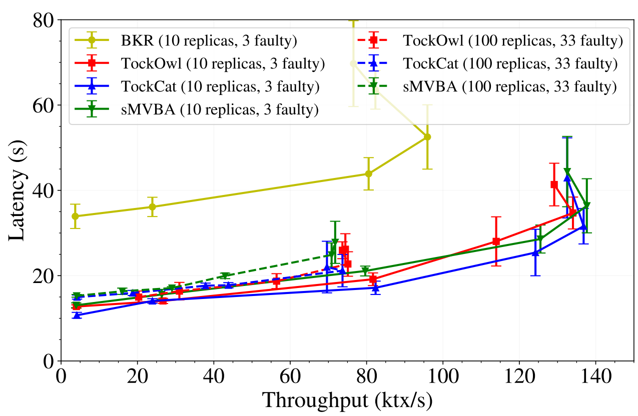


Figure 6: Performance of TockOwl under Byzantine faults.

its high communication and round complexity.

When faulty replicas are present, the performance of TockCat and sMVBA declines significantly, while TockOwl experiences only a slight decrease in performance. In this situation, TockOwl performs comparably to TockCat and sMVBA in a network of 10 replicas, but in a network of 100 replicas, TockOwl's performance is noticeably better than the other protocols. These results indicate that TockOwl is more stable in the presence of crash faults.

The performance decline of TockOwl is mainly affected by the bandwidth bottleneck of slower replicas among the surviving replicas, which can be resolved by increasing their resources. For consensus protocols like sMVBA, simply increasing resources cannot completely solve the problems caused by crashed replicas. More crashed replicas mean a greater probability that the common coin selects an ineffective leader. Once an ineffective leader is selected, even if the resources are sufficient, consensus cannot be reached in the current epoch.

Byzantine faults. We test the performance of the protocols in Byzantine environments. In this test, we disable the output shortcuts of TockCat and sMVBA. The strategies adopted by Byzantine replicas in the four protocols are as follows:

- TockOwl: Byzantine replicas only participate in the broadcast of the first phase. Under this strategy, if a Byzantine replica has the highest priority in the current epoch, consensus cannot be reached in that epoch.
- TockCat: Byzantine replicas only participate in the broadcast of the first phase and send empty *BestMsg* messages during the Best exchange step.
- sMVBA: Byzantine replicas only participate in the broadcast of the first phase and always vote 0 during the PreVote step.
- BKR: Byzantine replicas do not perform RBC and always vote 0 in ABA.

The results are shown in Figure 6. In a network of 10 replicas, BKR performs the worst, while the performance of the other three protocols is very close. In a network of 100 replicas, the performance of the three protocols remains very close, but TockCat performs slightly better than sMVBA and TockOwl. This aligns with the theoretical results, as TockCat has the least expected number of rounds. These results indicate that TockOwl's performance is not inferior to other protocols when facing Byzantine faults.

6.2 Tests on Network Adaptability

We test TockOwl+ and ParBFT in three environments: fast network (a good leader), slow network (a bad leader), and adversarial network (leader and replicas crash).

A good or bad leader. To intuitively show the impact of

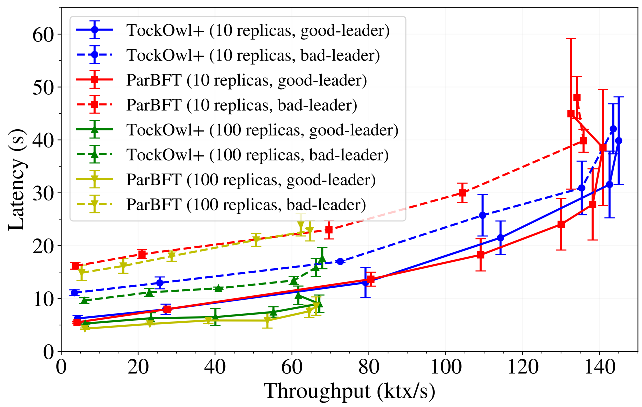


Figure 7: Performance of TockOwl+ under a good leader and a bad leader.

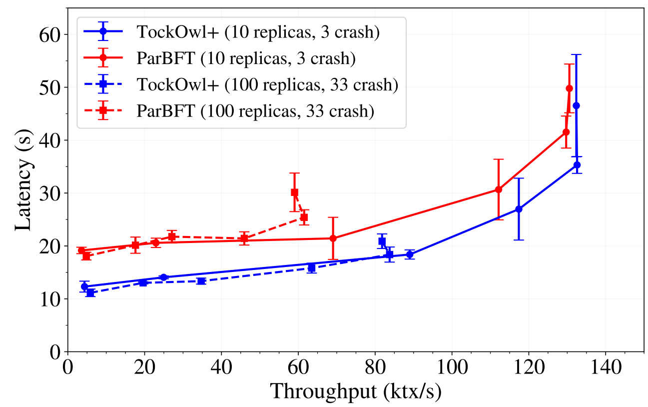


Figure 8: Performance of TockOwl+ under crash faults.

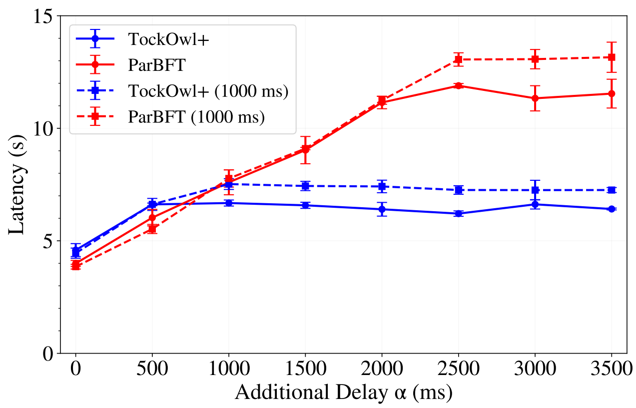


Figure 9: Performance when adding artificial network delay.

leader crashes on the protocols, we test a series of throughput and latency when the network size is 10 and 100. The results are shown in Figure 7. When the leader is fault-free, ParBFT performs slightly better than TockOwl+, mainly due to ParBFT's lower latency in optimistic scenarios. However, when the leader crashes, TockOwl+ performs significantly better than ParBFT, owing to TockOwl+'s concise slow track.

Leader and replica crash. To assess the protocols' performance in more adverse conditions, we test the protocols by crashing some non-leader replicas in addition to crashing the leader. The results are shown in Figure 8. It can be seen that, regardless of the network size, TockOwl+ performs significantly better than ParBFT. Furthermore, the performance gap between the two protocols widens compared to the scenario where only the leader fails. These results indicate that TockOwl+ performs better in slow and adversarial networks.

Impact of network delay. To further evaluate the effect of network latency on the protocols, we introduce an artificial network delay to the leader's messages, represented by the parameter α. This parameter can simulate scenarios where the leader experiences bandwidth limitations or is targeted by a denial-of-service attack, which is feasible since the leader's identity is typically public. We test the protocols under two patterns. In the first pattern, both tracks start synchronously, while in the second pattern, the slow track starts after a delay of Δ. In this experiment, the number of replicas is fixed at 100, each block contains 1000 transactions, and Δ is set to 1000 ms. We gradually change the value of α and obtain a series of latencies. The results are shown in Figure 9. The results show that as α increases, the latency of all protocols gradually rises. When α is less than 500 ms, the latency of TockOwl+ is dominated by the fast track. When α exceeds 500 ms, the latency of TockOwl+ transitions to being dominated by the slow track and remains stable thereafter. In contrast, ParBFT's latency does not stabilize until α reaches 2500 ms. Furthermore, the stable latency of ParBFT is significantly higher than that of TockOwl+.

7 Conclusion

This paper proposes an asynchronous BFT protocol called TockOwl which achieves crash robustness. TockOwl mainly consists of three steps: three-phase broadcast, common coin, and Best exchange. After the broadcast step, TockOwl utilizes a common coin to determine the priorities of replicas, rather than selecting a leader like traditional asynchronous BFT protocols. Furthermore, we introduce a protocol called TockOwl+, which is a dual-track protocol based on hedging delays. TockOwl+ has a faster slow track.

8 Acknowledgments

We thank the anonymous reviewers for their valuable comments, which greatly improved the quality of the paper. This paper is supported by the National Key R&D Program of China through project 2022YFB2702900, the Natural Science Foundation of China through projects U21A20467 and U24B20144 and 62272464.

Ethical Considerations and Open Science

Ethic Considerations. This paper presents no ethical risks, as all data used is publicly available. There are no concerns related to animals, human subjects, the environment, healthcare, or military applications. We confirm adherence to all ethical guidelines outlined in the CFPs.

Open Science This paper involves several consensus protocols. We have made the source code public, see `https://doi.org/10.5281/zenodo.14719752` and `https://github.com/yrdsm666/tockowl`.

References

[1] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In *PODC*, pages 381–391, 2022.

[2] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected o(1) rounds, expected communication, and optimal resilience. In *FC*, pages 320–334, 2019.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *SP*, pages 106–118, 2020.

[4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *PODC*, pages 337–346, 2019.

[5] The RabbitMQ author. Rabbitmq: One broker to queue them all, 2024.

[6] The Kubernetes authors. Production-grade container orchestration, 2024.

[7] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. Fnf-bft: A bft protocol with provable performance under attack. In *SIROCCO*, pages 165–198, 2023.

[8] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the limits of latency with uncertified dags. *arXiv preprint arXiv:2310.14821*, 2023.

[9] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. Spanner: Becoming a sql system. In *SIGMOD*, pages 331–343, 2017.

[10] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *PODC*, pages 183–192, 1994.

[11] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *PODC*, pages 183–192, 1994.

[12] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant cradient descent. *Advances in neural information processing systems*, 30, 2017.

[13] Erica Blum, Jonathan Katz, Julian Loss, Kartik Nayak, and Simon Ochsenreither. Abraxas: Throughput-efficient hybrid asynchronous consensus. In *CCS*, pages 519–533, 2023.

[14] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

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

[16] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, pages 524–541, 2001.

[17] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *SRDS*, pages 191–201, 2005.

[18] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, pages 173–186, 1999.

[19] Hao Cheng, Yuan Lu, Zhenliang Lu, Qiang Tang, Yuxuan Zhang, and Zhenfeng Zhang. Jumbo: Fully asynchronous bft consensus made truly scalable. *arXiv preprint arXiv:2403.11238*, 2024.

[20] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *TOCS*, 31(3):1–22, 2013.

[21] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

[22] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. Parbft: Faster asynchronous bft consensus with a parallel optimistic path. In *CCS*, pages 504–518, 2023.

[23] Francesco D'Amato, Joachim Neu, Ertem Nusret Tas, and David Tse. Goldfish: No more attacks on ethereum?! *Cryptology ePrint Archive*, 2022.

[24] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *EuroSys*, pages 34–50, 2022.

[25] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *SP*, pages 2502–2517, 2022.

[26] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *CCS*, pages 2705–2721, 2021.

[27] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *CCS*, pages 2028–2041, 2018.

[28] Sisi Duan, Xin Wang, and Haibin Zhang. Fin: Practical signature-free asynchronous common subset in constant time. In *CCS*, pages 815–829, 2023.

[29] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security*, pages 1605–1622, 2020.

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

[31] Bryan Ford. Threshold logical clocks for asynchronous distributed coordination and consensus. *arXiv preprint arXiv:1907.07010*, 2019.

[32] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *CCS*, page 1187–1201, 2022.

[33] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *FC*, pages 296–315, 2022.

[34] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. *arXiv preprint arXiv:2106.10362, version 2024-07-09*, 2024.

[35] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, pages 51–68, 2017.

[36] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. Beegees: stayin'alive in chained bft. In *PODC*, pages 233–243, 2023.

[37] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *DSN*, pages 568–580, 2019.

[38] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft bloser to practice. In *NDSS*, 2022.

[39] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *CCS*, pages 803–818, 2020.

[40] Lingling Guo, Qingfu Liu, Ke Shi, Yao Gao, Jia Luo, and Jingjing Chen. A blockchain-driven electronic contract management system for commodity procurement in electronic power industry. *Ieee Access*, 9:9473–9480, 2021.

[41] Bin Hu, Zongyang Zhang, Han Chen, You Zhou, Huazu Jiang, and Jianwei Liu. Dycaps: Asynchronous proactive secret sharing for dynamic committees. *IACR Cryptol. ePrint Arch.*, 2022(1169), 2022.

[42] Jian Hu, Peng Zhu, Yong Qi, Qingyun Zhu, and Xiaotong Li. A patent registration and trading system based on blockchain. *Expert Systems with Applications*, 201:117094, 2022.

[43] Kexin Hu, Kaiwen Guo, Qiang Tang, Zhenfeng Zhang, Hao Cheng, and Zhiyang Zhao. Leopard: Towards high throughput-preserving bft for large-scale systems. In *ICDCS*, pages 157–167, 2022.

[44] Nan Jing, Qi Liu, and Vijayan Sugumaran. A blockchain-based code copyright management system. *Information Processing & Management*, 58(3):102518, 2021.

[45] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *PODC*, pages 165–175, 2021.

[46] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *PODC*, pages 165–175, 2021.

[47] Aleksandar Kircanski and Terence Tarvis. Coinbugs: enumerating common blockchain implementation-level vulnerabilities. *arXiv preprint arXiv:2104.06540*, 2021.

[48] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, pages 45–58, 2007.

[49] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[50] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

[51] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[52] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[53] Minghang Li, Qianhong Wu, Zhipeng Wang, Bo Qin, Bohang Wei, Hang Ruan, Shihong Xiong, and Zhenyang Ding. Tockowl: Asynchronous consensus with fault and network adaptability. 2025.

[54] Tian Li, Shengyuan Hu, Ahmad Beirami, and Virginia Smith. Ditto: Fair and rrobust federated learning through personalization. In *ICML*, pages 6357–6368, 2021.

[55] Chao Liu, Sisi Duan, and Haibin Zhang. Epic: Efficient asynchronous bft with adaptive security. In *DSN*, pages 437–451, 2020.

[56] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. Xft: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.

[57] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. Flexible advancement in asynchronous bft consensus. In *SOSP*, pages 264–280, 2023.

[58] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *CCS*, pages 887–903, 2019.

[59] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft. In *CCS*, pages 2159–2173, 2022.

[60] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *PODC*, pages 129–138, 2020.

[61] Feng Ma, Ning Tang, Rui Xu, and Ziqian Zhang. Electronic contract ledger system based on blockchain technology. In *Journal of Physics: Conference Series*, volume 1828, page 012112, 2021.

[62] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *CCS*, pages 1041–1053, 2019.

[63] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *CCS*, pages 31–42, 2016.

[64] Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *CCS*, pages 1686–1699, 2021.

[65] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Randomized intrusion-tolerant asynchronous services. In *DSN*, pages 568–577, 2006.

[66] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t< n/3 and o (n2) messages. In *PODC*, pages 2–9, 2014.

[67] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–319, 2014.

[68] HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *OPODIS*, pages 88–102, 2005.

[69] Matthieu Rambaud. Faster asynchronous blockchain consensus and mvba. *Cryptology ePrint Archive*, 2024.

[70] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *CSUR*, 22(4):299–319, 1990.

[71] Zohar A Sompolinsky Y. Secure high-rate transaction processing in bitcoin. In *FC*, pages 507–527, 2015.

[72] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *CCS*, pages 2705–2718, 2022.

[73] Xiao Sui, Sisi Duan, and Haibin Zhang. Marlin: Two-phase bft with linearity. In *DSN*, pages 54–66, 2022.

[74] Amit Taneja, M Sankar, Shaik Vaseem Akram, D Praveenadevi, Hemant Singh Pokhariya, et al. A sizeable empirical investigation of bug attributes in blockchain structures. In *ICACITE*, pages 1872–1877, 2023.

[75] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. Quepaxa: Escaping the tyranny of timeouts in consensus. In *SOSP*, pages 281–297, 2023.

[76] Khin Me Me Thein. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[77] Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Strengthened fault tolerance in byzantine fault tolerant replication. In *ICDCS*, pages 205–215, 2021.

[78] Yang Xiao, Ning Zhang, Wenjing Lou, and Y Thomas Hou. A decentralized truth discovery approach to the blockchain oracle problem. In *INFOCOM*, pages 1–10, 2023.

[79] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. Dispersedledger:high-throughput byzantine consensus on variable bandwidth networks. In *NSDI*, pages 493–512, 2022.

[80] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *PODC*, pages 347–356, 2019.

[81] Haibin Zhang and Sisi Duan. Pace: Fully parallelizable bft from reproposable byzantine agreement. In *CCS*, page 3151–3164, 2022.

[82] Haibin Zhang, Sisi Duan, Boxin Zhao, and Liehuang Zhu. Waterbear: Practical asynchronous bft matching security guarantees of partially synchronous bft. In *USENIX Security*, pages 5341–5357, 2023.

[83] Zongyang Zhang, You Zhou, Sisi Duan, Haibin Zhang, Bin Hu, Licheng Wang, and Jianwei Liu. Dory: Faster asynchronous bft with reduced communication for permissioned blockchains. *Cryptology ePrint Archive*, 2022.

A Correctness Proof of TockOwl+

Lemma 7. *If correct replicas p_i and p_j commit $Proposal1$ and $Proposal2$ with epoch number e in epoch e, respectively, then $Proposal1 = Proposal2$.*

Proof. We consider three cases:

- **Case1:** Both p_i and p_j commit a proposal in the fast track. Since correct replicas do not vote for different proposals in the fast track, the proposals committed by correct replicas are consistent.
- **Case2:** Both p_i and p_j commit a proposal in the slow track. Due to the safety of TockOwl, we can directly infer the safety of this case.
- **Case3:** p_i commits a proposal in the fast track, while p_j commits a proposal in the slow track. We denote the leader as p_l, and its proposal and corresponding three-phase QCs as $proposal_l$, $qc1_l$, $qc2_l$, and $qc3_l$.
  - If p_i first commits $proposal_l$ in the fast track, it means that at least $n-2f$ correct replicas have voted for $qc2_l$ and stored $qc2_l$ in their $Q2$ set. Then, in the Best exchange step, all correct replicas will store $qc2_l$ in their $Q2$ set. Since the leader has the highest priority, there will not exist a proposal different from $proposal_l$ can be committed in the slow track.
  - If p_j first commits a proposal in the slow track, it means that at least $n-2f$ correct replicas have broadcast $BestMsg$ messages and stopped voting in the three-phase broadcast. At this point, the fast track will not be able to make progress, and other correct replicas cannot commit another proposal in the fast track.

□

Theorem 3 (Safety)**.** *In TockOwl+, if a correct replica p_i commits $Proposal1$ with epoch number e in epoch e, and a correct replica p_j commits $Proposal2$ with epoch number e' in epoch e', then either $Proposal1$ extends $Proposal2$ or $Proposal2$ extends $Proposal1$.*

Proof. When $e = e'$, the proof of the lemma can be directly derived from Lemma 7. Without loss of generality, we assume $e < e'$. In epoch e, we consider two cases:

- **Case1:** p_i commits $Proposal1$ in the slow track. In this case, the lemma can be directly derived from Theorem 1 and its proof.
- **Case2:** p_i commits $Proposal1$ in the fast track. Let p_l be the leader of epoch e, let the corresponding three-phase QCs be $qc1_l$, $qc2_l$, and $qc3_l$, let $priority$ be the priority of p_l in epoch e. At this point, any correct replica sets its own $parentQc1 = qc1_l$, and $parentQc2 = qc2_l$. In epoch e, there does not exist a first-phase QC with a priority greater than $priority$ because the leader's priority is the highest. In epoch $e+1$, if a Byzantine replica references a proposal with a priority lower than $priority$, then $\texttt{Pri}_e(proposal.parentQc) < priority = \texttt{Pri}_e(qc2_l)$ holds. This means that this proposal will not pass the SAFEPROPOSAL check in Alg. 2, making the proposal invalid. In other words, in epoch $e+1$, all valid proposals will reference $Proposal1$, so the subsequently committed proposals will extend from $Proposal1$.

□

Theorem 4 (Liveness)**.** *Each correct replica eventually commits some proposals.*

Proof. We consider two cases:

- **Case1:** There exist some correct replicas, such as p_i, that commit the leader's proposal in the fast track. If a correct replica p_j cannot commit in the fast track, then eventually it will trigger a timeout and send messages about the slow track to all replicas. Upon receiving the message from p_j, p_i will forward the $HaltMsg$ to p_j to help it commit. Therefore, every correct replica can eventually commit some proposals.
- **Case2:** If no correct replica commits a proposal in the fast track, then eventually all correct replicas will start the slow track. Since TockOwl has liveness, all correct replicas will eventually commit some proposals in the slow track.

□

B TockCat: Asynchronous BFT SMR with low latency

In this section, we present TockCat, an asynchronous BFT SMR protocol that achieves quadratic communication complexity with an expected latency of 10.5 rounds[5]. TockCat ranks among the fastest asynchronous multi-value BFT protocols in terms of round efficiency.

Algorithm 4 Two-phase broadcast for TockCat (for epoch e, replica p_i)

Global: $V, Q1, Q2$
Input: $proposal_i$

```
    // Main Broadcast Process
    input (cbc1, proposal_i, null) to CBC1_i
    upon outputting (cbc1, proposal_i, qc1_i) from CBC1_i do
        input (cbc2, H(proposal_i), qc1_i) to CBC2_i
    upon outputting (cbc2, H(proposal_i), qc2_i) from CBC2_i do
        broadcast (last, H(proposal_i), qc2_i)

    upon receiving (cbc1, proposal_j, null) from p_j do
        if SAFEPROPOSAL(proposal_j) then // every replica verifies
    whether this rule holds before participating in CBC1_j
            add proposal_j to V
    upon receiving (cbc2, H(proposal_j), qc1_j) from p_j do
        add qc1_j to Q1
    upon receiving (last, H(proposal_j), qc2_j) from p_j do
        add qc2_j to Q2
    upon |V|, |Q1|, |Q2| are all not less than n - f do
        Output Finish
```

B.1 Protocol Description

The complete description of TockCat is shown in Alg. 4 and 5. TockCat operates across epochs, during which replicas propose a new proposal in each epoch. We assume that all message formats are authentic and legitimate, and originate from the current epoch, denoted as e.

Two-phase broadcast. TockCat employs a two-phase broadcast instead of a three-phase broadcast. During this step, proposals received by a replica are stored in the set V, while the QCs from the first and second phases are stored in the sets $Q1$ and $Q2$, respectively. The replica also performs a SAFEPROPOSAL check on the proposals from other replicas (Alg. 5, Line 3). The main purpose of this check is to validate the proof contained in the proposal. If a proposal references the leader's proposal from the previous epoch, the proof should be the leader's first-phase QC. Conversely, if the proposal references the replica's own proposal from the previous epoch, the proof should be $\sigma_{e-1,non-leader}$ (see below). If the SAFEPROPOSAL check fails, the replica refuses to vote.

Common coin. After completing the two-phase broadcast, the replica broadcasts a share of the common coin and waits

[5]The expected number of rounds is calculated using the same method as Speeding-Dumbo [38], which accounts for the complete rounds of an epoch. 2PAC [69] considers the benefit provided by the output shortcut, which reduces the round count by one. If we apply this calculation method, TockCat's expected latency would be 9.5 rounds.

Algorithm 5 TockCat protocol (for epoch e, replica p_i)

Initialization: If $e = 1$, then $parentHash, parentProof \leftarrow null$. $V, Q1, Q2 \leftarrow \{\}$. Let *value* represent the value input by p_i.

// Utilities

procedure UPDATEPARENT($proposal$, $proof$)
  $parentHash \leftarrow H(proposal)$, $parentProof \leftarrow proof$

procedure SAFEPROPOSAL($proposal(e, value, hash, proof)$)
  Verify the signature in $proof$ according to $hash$

// Two-phase broadcast

$proposal \leftarrow (e, value, parentHash, parentProof)$
input *proposal* to Two-phase broadcast
upon outputting `Finish` from Two-phase broadcast **do**
  if p_i has not broadcast *ShareMsg* message **then**
    broadcast *ShareMsg*(*coinShare*)

// Common Coin

upon receiving *ShareMsg* messages from $n-2f$ replicas **do**
  if p_i has not broadcast *ShareMsg* message **then**
    broadcast *ShareMsg*(*coinShare*)
wait until receiving *ShareMsg* messages from $n-f$ replicas
  $lqc1, lqc2, s \leftarrow null$
  $l \leftarrow LeaderElection(e)$ // p_l is the leader of epoch e
  stop participating in unfinished CBC
  if $qc2_l$ is in $Q2$ **then**
    $lqc2 \leftarrow qc2_l$
  if $qc1_l$ is in $Q1$ **then**
    $lqc1 \leftarrow qc1_l$
  else
    $s \leftarrow$ the signature share for $(e, non-leader)$
  broadcast $BestMsg(lqc1, lqc2, s)$

// Best exchange

upon receiving $BestMsg(lqc1, lqc2, s)$ from p_j **do**
  add $lqc1, lqc2$ to $Q1, Q2$, respectively
wait until receiving *BestMsg* messages from $n-f$ replicas
  if $qc1_l$ is in $Q1$ **then**
    UPDATEPARENT($proposal_l$, $qc1_l$)
  else
    combine $n-f$ signature shares s and obtain the threshold signature $\sigma_{e,non-leader}$
    UPDATEPARENT($proposal_i$, $\sigma_{e,non-leader}$)

// Commit

upon observing $qc2_l$ is in $Q2$ at any time **do**
  Commit $proposal_l$ and its uncommitted ancestor proposals

for the shares from other replicas. In this process, we set the coin threshold to $n-f$. If a replica receives $n-2f$ shares and has not yet broadcast its own share, it then broadcasts its share. Eventually, each replica receives $n-f$ shares, then it computes the coin value and selects a leader.

Best exchange. If the replica finds the leader's $qc1_l$ in its own Q1 set, it broadcasts a *BestMsg* message containing $qc1_l$ and $qc2_l$ (if available). Otherwise, the replica generates a signature share s for $(e, non-leader)$ and broadcasts a *BestMsg* message containing s.

The replica waits until it receives $n-f$ *BestMsg* messages. If at least one of these contains $qc1_l$, the replica sets $parentHash = H(proposal_l)$ and $parentProof = qc1_1$, then moves to the next epoch. Otherwise, the replica combines the $n-f$ signature shares to obtain the threshold signature $\sigma_{e,non-leader}$, which proves that $n-f$ replicas have generated a signature share for $(e, non-leader)$. In this case, the replica sets $parentHash = H(proposal_i)$, $parentProof = \sigma_{e,non-leader}$, and enters the next epoch.

Once the replica observes $qc2_l$, it can immediately commit the corresponding $proposal_l$. This condition may be triggered when the coin value is revealed or when receiving *BestMsg* messages.

B.2 Protocol Correctness

The proof of TockCat's liveness is similar to that of TockOwl, so it is not repeated here. Instead, we focus on providing the safety proof for TockCat. If $Proposal1$ is an ancestor proposal of $Proposal2$, we say that $Proposal2$ extends $Proposal1$.

Theorem 5 (Safety)**.** *In TockCat, if a correct replica p_i commits $Proposal1$ with epoch number e in epoch e, and a correct replica p_j commits $Proposal2$ with epoch number e' in epoch e', then either $Proposal1$ extends $Proposal2$ or $Proposal2$ extends $Proposal1$.*

Proof. If $e = e'$, then $Proposal1 = Proposal2$, because p_i and p_j can only commit the leader's proposal in the current epoch.

Without loss of generality, we assume $e < e'$. In epoch e, let p_l be the leader. Replica p_i can only commit p_l's proposal after obtaining $qc2_l$. This indicates that at least $n-2f$ correct replicas have voted for $qc1_l$ and added $qc1_l$ to their respective $Q1$ set. These correct replicas broadcast *BestMsg* messages containing $qc1_l$ during the Best exchange step. Due to quorum intersection, any correct replica p_k will receive $qc1_l$ and set $parentHash = H(Proposal1)$ and $parentProof = qc1_l$. Moreover, no replica can produce a threshold signature $\sigma_{e,non-leader}$ for $(e, non-leader)$. Therefore, in subsequent epochs, all valid proposals can only extend the proposal corresponding to $qc1_l$, which is $Proposal1$. □

Lemma 8 (Efficiency)**.** *TockCat has a communication complexity of $O(n^2)$ and an expected latency of 10.5 rounds.*

Proof. The protocol uses all-to-all broadcasting in each round, so it has $O(n^2)$ communication complexity. The protocol requires 7 rounds per epoch, and the probability of committing in each epoch is 2/3, leading to an expected latency of 10.5 rounds. □

Note that if we consider the gain brought by the commit shortcut (triggered when the common coin value is revealed), the probability of TockCat committing a proposal in 6 rounds (1 epoch) is 2/3, the probability in $7+6$ rounds (2 epochs) is $1/3 \times 2/3$, and so on. This results in an expected latency of 9.5 rounds for TockCat, which is comparable to the current fastest asynchronous multi-value Byzantine consensus protocol, 2PAC [69]. Before 2PAC, the fastest protocol was Ditto [33]. However, the initial version of Ditto had a safety flaw, which was fixed by 2PAC. The latest version of Ditto [34] uses a MVBA protocol as a black box, and when using 2PAC, Ditto's latency is 13.5 rounds.