Wahoo: A DAG-Based BFT Consensus With Low Latency and Low Communication Overhead

Xiaohai Dai, *Member, IEEE*, Zhaonan Zhang, *Student Member, IEEE*, Zhengxuan Guo, *Student Member, IEEE*, Chaozheng Ding, *Student Member, IEEE*, Jiang Xiao, *Member, IEEE*, Xia Xie, *Member, IEEE*, Rui Hao, *Member, IEEE*, and Hai Jin, *Fellow, IEEE*

***Abstract*— To parallelize data processing within BFT consensus protocols, *Directed Acyclic Graph* (DAG) structures have been integrated into consensus design, shaping the realm of DAG-based BFT protocols. Existing DAG-based protocols rely on the *Reliable Broadcast* (RBC) protocol or its variants for block dissemination, which ensures consistency and totality properties of the data delivery. However, the inherent communication overhead of $O(n^2)$ in RBC (where n is the total replica count) results in an unwieldy $O(n^3)$ overhead in current DAG-based solutions, as each replica disseminates blocks through RBC in parallel. In response to this issue, we propose two new broadcast protocols: *Provable Broadcast* (PBC) and *Enhanced Provable Broadcast* (EPBC). Both PBC and EPBC maintain the consistency property of data delivery, similar to RBC, while offering linear communication overhead without totality. Leveraging these broadcast protocols, we devise Wahoo, a novel DAG-based BFT protocol that significantly reduces communication overhead to $O(n^2)$. To address the absence of the totality property, we introduce a block retrieval mechanism to assist replicas in acquiring missing blocks. Additionally, under favorable conditions, Wahoo achieves a low latency of 4δ (where δ symbolizes the actual network delay), rivaling the best performance of existing DAG-based protocols. Various experiments showcase Wahoo's high performance, owing to its substantially reduced communication overhead.**

***Index Terms*— *Byzantine Fault Tolerant* (BFT), consensus, *Directed Acyclic Graph* (DAG), blockchain.**

I. Introduction

With the burgeoning popularity of blockchain technology [5], [27], the *Byzantine Fault Tolerant* (BFT) consensus has garnered renewed attention, serving as a pivotal foundation for blockchain systems [43], [44]. At a high level, the BFT consensus protocol facilitates agreement on data among a distributed group of replicas, some of which act as Byzantine replicas, deviating arbitrarily from the protocol. Traditional BFT consensus protocols, like PBFT [13] and HotStuff [46], process data in a sequential manner, significantly constraining the system throughput.

To enhance system throughput, a line of recent studies, such as DAGRider [28] and GradedDAG [45], have introduced the *Directed Acyclic Graph* (DAG) structure to consensus design, parallelizing data processing. These DAG-based BFT consensus protocols operate in successive rounds, with each replica broadcasting a block through the *Consistent Broadcast* (CBC) [41], *Reliable Broadcast* (RBC) [9], or a variant called GRBC [45] protocol in each round. Both CBC and (G)RBC can guarantee the consistency property of data delivery, and (G)RBC can further ensure the totality property. However, both CBC and (G)RBC incur a communication overhead of $O(n^2)$, where n is the number of replicas, resulting in an overall communication overhead of $O(n^3)$ for DAG-based consensus.

In this paper, we aim to propose a new DAG-based BFT consensus protocol to reduce communication overhead. Through an analysis of DAG-based protocols, we discover that CBC or (G)RBC is not indispensable in designing DAG-based protocols. Instead, we introduce two new broadcast protocols, namely *Provable Broadcast* (PBC) and *Enhanced Provable Broadcast* (EPBC), both boasting linear communication overhead. While PBC and EPBC guarantee the consistency property, akin to CBC or (G)RBC, they forego the totality property.

PBC can be simply implemented within three communication steps, while EPBC involves five communication steps, whose delivery includes three distinctive tags: $\mathtt{T}_{S1}$, $\mathtt{T}_{S2}$, and $\mathtt{T}_{F}$. In EPBC, a replica delivering data with the $\mathtt{T}_{S2}$ tag can assert that a sufficient number of non-faulty replicas have delivered the same data with the $\mathtt{T}_{S1}$ tag. Similarly, a replica delivering data with the $\mathtt{T}_{F}$ tag can assert that enough non-faulty replicas have received the same data. These two assertions play a pivotal role in upholding consensus safety. A replica in EPBC can deliver data with $\mathtt{T}_{S1}$ or $\mathtt{T}_{S2}$ after three or five communication steps, respectively. Besides, in favorable situations, it can deliver data with $\mathtt{T}_{F}$ after three steps.

Manuscript received 1 January 2024; revised 2 April 2024; accepted 27 May 2024. Date of publication 3 June 2024; date of current version 14 August 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB2700700 and in part by the Key Research and Development Program of Hubei Province under Grant 2021BEA164. The associate editor coordinating the review of this article and approving it for publication was Prof. Haibo Hu. *(Corresponding author: Jiang Xiao.)*

Xiaohai Dai, Zhaonan Zhang, Zhengxuan Guo, Chaozheng Ding, Jiang Xiao, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Laboratory, and the Cluster and Grid Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: xhdai@hust.edu.cn; zhaonanzhang@hust.edu.cn; gzxhere@hust.edu.cn; chaozhengding@hust.edu.cn; jiangxiao@hust.edu.cn; hjin@hust.edu.cn).

Xia Xie is with the School of Computer Science and Technology, Hainan University, Haikou 570228, China (e-mail: shelicy@hainanu.edu.cn).

Rui Hao is with the School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan 430070, China (e-mail: ruihao@whut.edu.cn).

Digital Object Identifier 10.1109/TIFS.2024.3409082

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

Leveraging PBC and EPBC, we devise Wahoo, a DAG-based BFT protocol with $O(n^2)$ communication overhead—an order of magnitude less than existing solutions. Wahoo operates in successive waves, with each wave comprising an EPBC phase and a PBC phase. In the EPBC phase, each replica broadcasts a block through EPBC, named the EPBC block. An EPBC block must include at least n-f hashes of blocks delivered in the previous PBC phase, where f denotes the fault tolerance. After delivering n-f EPBC blocks, a replica transitions to the PBC phase, broadcasting a block through PBC, named the PBC block. A PBC block also has to include at least n-f hashes of blocks delivered in the previous EPBC phase, regardless of the delivery tag.

Each PBC block includes a threshold signature share on the wave number, a mechanism used to randomly select a leader block for the wave. This mechanism, also known as *Global Perfect Coin* (GPC), designates an EPBC block within this wave as the leader block. If the leader block is delivered with the $\mathtt{T_{S2}}$ or $\mathtt{T_F}$ tag, it and its ancestor blocks can be directly committed. In favorable scenarios where n-f EPBC blocks are delivered with $\mathtt{T_F}$, each replica advances to the PBC phase after only three communication steps. The first communication step of the PBC phase can reveal the leader block. In such scenarios, a leader block can be committed in just four steps, resulting in a low consensus latency of 4δ, equivalent to the best result achieved by GradedDAG.

Even in unfavorable scenarios, where some EPBC blocks are delivered with the tag $\mathtt{T_{S2}}$, each replica advances to the PBC phase after five communication steps. The probability of the elected leader block being delivered with tag $\mathtt{T_{S2}}$ or $\mathtt{T_F}$ is approximately 2/3. Therefore, in expectation, it takes about 1.5 waves to commit a leader block. Given that each wave consists of eight communication steps in unfavorable scenarios, Wahoo's expected latency is approximately 12δ. Regardless of the scenario, Wahoo employs n parallel EPBC or PBC instances to broadcast blocks, resulting in $O(n^2)$ communication overhead. To supplement the totality property absent in PBC and EPBC, we introduce a block retrieval mechanism, assisting replicas in acquiring missing blocks.

We conduct a thorough analysis of Wahoo's correctness, demonstrating its ability to satisfy both safety and liveness properties. A series of experiments are conducted to verify Wahoo's efficiency. Tusk and GradedDAG, representing the state-of-the-art, are chosen as comparison counterparts. We compare different protocols' performance under both favorable and unfavorable situations. Experimental results reveal that Wahoo outperforms others, especially in scenarios with a large number of replicas. This superior performance is attributed to Wahoo's lower communication overhead.

In summary, this paper makes the following contributions:

- To reduce quadratic communication overhead found in the broadcast primitives of existing DAG-based BFT protocols, we propose two new broadcast protocols named PBC and EPBC, which achieve linear overhead.
- Based on PBC and EPBC, we devise Wahoo, a novel DAG-based BFT protocol with $O(n^2)$ communication overhead, which delivers low latency of 4δ under favorable situations, comparable to state-of-the-art protocols.
- We conduct several sets of experiments to evaluate Wahoo, whose results validate its high efficiency and low communication overhead.

The rest of this paper is organized as follows. We first provide background and preliminary knowledge in Section II. After defining the models and introducing some building blocks in Section III, we elaborate on the design of Wahoo in Section IV. Section V analyzes its correctness, while Section VI verifies the protocol's efficiency through various experiments. We finally make a summary of related works in Section VII and conclude the paper in Section VIII.

II. Background & Preliminaries

In this section, we provide essential background information on BFT consensus and the asynchronous DAG-based consensus, which motivates our work in this paper.

A. BFT Consensus and Timing Assumptions

As a fundamental component of the blockchain system [5], [27], BFT consensus [43], [44] has regained substantial attention over the past decade, owing to the explosive rise of blockchain technology. Generally speaking, BFT consensus aims to facilitate agreement on data among distributed replicas, when a subset of these replicas behaves arbitrarily, often referred to as Byzantine replicas [33].

In the context of blockchain technology, all committed data are structured into units called blocks, with each committed block assigned a sequence number to indicate its position in the chain. A block consists of multiple transactions, which are generated and submitted by upper-layer clients. A BFT consensus protocol has to satisfy the following three properties:

- **Safety:** If two non-faulty replicas p_i and p_j commit two blocks B_i and B_j with the same sequence number, B_i must be equal to B_j.
- **Liveness:** If a client submits a transaction tx to the system, tx will eventually be committed.
- **Completeness:** For each sequence number k, each non-faulty replica will commit a block numbered k eventually.

The network assumptions play a crucial role in designing BFT consensus protocols and can be categorized into three types: synchronous, partially-synchronous, and asynchronous [19]. The synchronous network assumes that all messages can be delivered within a predetermined period, while the asynchronous network places no constraints on delivery timing. The partially-synchronous network, serving as an intermediary, assumes a synchronous network after an unknown *Global Stabilization Time* (GST). Consequently, BFT consensus protocols fall into three categories. While synchronous and partially-synchronous protocols promise higher performance, they are criticized for vulnerability to network attacks [25], [38]. Recent studies have thus refocused on asynchronous protocols and aimed to improve the asynchronous protocol's practicability [18], [22], [47].

B. DAG-Based Consensus

Traditional BFT protocols, such as PBFT [13] and Zyzzyva [32], process transactions individually. The advent

TABLE I
COMPARISON BETWEEN DIFFERENT DAG-BASED PROTOCOLS. LATENCY IS MEASURED BY THE COMMUNICATION STEPS

	Wave length	Broadcast protocol	Good-case latency[†]	Expected worst latency[‡]	Communication overhead
DAGRider [28]	4	RBC	12 (10)	18	$O(n^3)$
Tusk [16]	3	RBC	9 (7)	21	$O(n^3)$
BullShark [40]	4	RBC	6	30	$O(n^3)$
GradedDAG [45]	2	GRBC & CBC	5 (4)	7.5	$O(n^3)$
Wahoo (This work)	2	EPBC & PBC	6 (4)	12	$O(n^2)$

[†] If the correctness is not compromised, our focus lies solely on counting the initial step in RBC, CBC, or PBC that reveals the leader block. This approach serves to minimize latency, as indicated by the results enclosed in brackets.
[‡] The term 'expected worst latency' denotes the expected latency in situations where the adversary launches attacks arbitrarily.

of blockchain technology introduced a novel approach to organizing transactions as blocks and linking all blocks into a chain, which simplifies the consensus design with notable examples being Tendermint [10] and HotStuff [46]. However, these chain-based protocols process data sequentially, resulting in a significant limitation in throughput. To address this, the DAG structure is introduced to consensus design, giving rise to DAG-based consensus [16], [28], [40], [45]. To guarantee liveness, these DAG-based protocols adopt the assumption of an asynchronous network. However, they rely on the RBC protocol or its variant, *Graded RBC* (GRBC), to broadcast blocks. Given the quadratic communication overhead of (G)RBC, consensus protocols built upon it suffer from high communication overhead, often reaching cubic levels, as illustrated in Table I.

This prompts a natural question: *Can we design a DAG-based BFT consensus protocol with lower communication overhead?*

C. Global Perfect Coin (GPC)

In response to the FLP impossibility [20], asynchronous consensus protocols [3], [6], [35] must incorporate randomness, typically achieved through the GPC. The GPC works like a global function; when a specified threshold of replicas triggers it with the same input, it consistently produces an output. Crucially, GPC maintains the following properties:

- **Consistency:** If two non-faulty replicas receive outputs o and o' from the same input, then $o = o'$.
- **Unpredictability:** With at most $t-1$ replicas triggering the GPC function with an input (where t is the threshold), no replica can forecast the output.
- **Termination:** When at least t replicas engage GPC with the same input, all replicas can eventually output.
- **Randomness:** GPC outputs uniformly random results.

GPC can be implemented easily through threshold signatures [8], [12], configuring the threshold parameter as t. Replicas initiate GPC by generating and broadcasting a signature share based on the input. Upon collecting t shares, a replica constructs a complete threshold signature, symbolizing GPC's output. In DAG-based BFT scenarios, the GPC function plays a critical role in selecting leaders for waves.

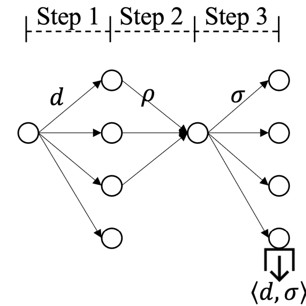


Fig. 1. The algorithm construction of PBC ($\mathtt{ALG}_{pbc}$).

III. MODELS AND BUILDING BLOCKS

In this section, we outline the models underpinning our protocol and introduce essential building blocks integral to our design.

A. Models and Definitions

The system is composed of n replicas, allowing for a maximum of f Byzantine replicas (where $n \geq 3f+1$), which achieves optimal resilience. Each replica is uniquely identified by the notation p_i $(0 \leq i < n)$. Byzantine replicas fall under the manipulation of an adversary capable of coordinating their actions. Every pair of replicas is interconnected through a reliable network link. The network is assumed to be asynchronous, with no specific assumptions regarding message delivery timings. The adversary possesses the capability to delay messages arbitrarily, as long as these messages can eventually be delivered.

Within this system, a robust security framework is established, incorporating a *Public Key Infrastructure* (PKI) and a *Threshold Signature Infrastructure* (TSI). PKI facilitates global verification of a message sender's identity, immune to falsification or repudiation. TSI implementation involves a trusted leader or a *Distrusted Key Generation* (DKG) protocol [2], [17], [31]. The assumption of computational limitations restricts the adversary from compromising the safety of PKI or TSI. The threshold t for GPC is set to $f+1$.

B. Building Block: PBC

The *Provable Broadcast* (PBC) protocol functions as a stronger broadcast primitive, which ensures non-faulty replicas

deliver identical data (if capable) despite potential Byzantine behavior from the broadcaster. In addition to delivering data, a replica in PBC also delivers a certificate for the data, serving as proof for the data delivery. A global validation predicate, denoted as Q, verifies this proof. Specifically, a non-faulty replica delivers a tuple from PBC in the format $\langle d, \sigma \rangle$, where d signifies the data and σ represents the certificate, and $Q(d, \sigma) = \text{True}$. The PBC protocol, for a single broadcast instance, adheres to the following properties:

- **Consistency.** In a PBC instance, if two non-faulty replicas deliver $\langle d, \sigma \rangle$ and $\langle d', \sigma' \rangle$ respectively, then $d = d'$ and $\sigma = \sigma'$.
- **Validity.** When the broadcaster is non-faulty and transmits data d, each replica will deliver $\langle d, \sigma \rangle$, where $Q(d, \sigma)$=True.
- **Integrity.** Each non-faulty replica delivers at most once in a PBC instance.

PBC can be constructed through three communication steps, represented as $\texttt{ALG}_{pbc}$ and demonstrated in Fig. 1. In the first step, the broadcaster transmits its data d to each replica. Upon receiving d, a replica generates a (n-f)-threshold signature share ρ on d and returns it to the broadcaster. After collecting n-f shares, the broadcaster generates a complete threshold signature σ and broadcasts it in the third step. Upon receiving σ, a replica delivers $\langle d, \sigma \rangle$. The validation predicate Q is set to the (n-f)-threshold signature verification function. It is easy to find that $\texttt{ALG}_{pbc}$ has a latency of 3δ and linear communication overhead, where δ denotes the actual network delay.

C. Building Block: EPBC

1) Definition: The *Enhanced Provable Broadcast* (EPBC) protocol is an enhanced version of PBC. A replica delivers a three-element tuple $\langle d, t, \sigma \rangle$ if capable, where d and σ retain the same definitions as PBC, and t ($t \in \{\texttt{T}_{\texttt{S1}}, \texttt{T}_{\texttt{S2}}, \texttt{T}_{\texttt{F}}\}$) denotes a delivery tag. On the one hand, if a non-faulty replica delivers $\langle d, \texttt{T}_{\texttt{S2}}, \sigma_{\texttt{S2}} \rangle$, it can assert that at least $n - 2f$ non-faulty replicas have delivered $\langle d, \texttt{T}_{\texttt{S1}}, \sigma_{\texttt{S1}} \rangle$. On the other hand, if a non-faulty replica delivers $\langle d, \texttt{T}_{\texttt{F}}, \sigma_{\texttt{F}} \rangle$, it can assert that all non-faulty replicas have received d. It is important to note that 'deliver' holds more significance than 'receive'. After a replica receives some data, it can deliver the data only if some stronger condition is met. Two global validation predicates, denoted as Q and P, are defined to verify the delivered tuple's correctness. For a single instance of broadcast, EPBC must satisfy the following properties:

- **Consistency.** In an EPBC instance, if two non-faulty replicas deliver $\langle d, t, \sigma \rangle$ and $\langle d', t', \sigma' \rangle$, then $d = d'$.
- **Validity.** If the broadcaster is non-faulty and broadcasts d, each non-faulty replica will deliver $\langle d, \texttt{T}_{\texttt{S1}}, \sigma_{\texttt{S1}} \rangle$ and $\langle d, \texttt{T}_{\texttt{S2}}, \sigma_{\texttt{S2}} \rangle$, where $Q(d, \texttt{T}_{\texttt{S1}}, \sigma_{\texttt{S1}}) = \text{True}$ and $Q(d, \texttt{T}_{\texttt{S2}}, \sigma_{\texttt{S2}}) = \text{True}$. Besides, if all replicas are non-faulty, each replica will deliver $\langle d, \texttt{T}_{\texttt{F}}, \sigma_{\texttt{F}} \rangle$, where $P(d, \texttt{T}_{\texttt{F}}, \sigma_{\texttt{F}}) = \text{True}$.
- **Integrity.** Each non-faulty replica can deliver at most three times in an EPBC instance.
- **Delivery assertion.** If a non-faulty replica delivers $\langle d, \texttt{T}_{\texttt{S2}}, \sigma_{\texttt{S2}} \rangle$ in an EPBC instance, at least $n - 2f$ non-faulty replicas must have delivered $\langle d, \texttt{T}_{\texttt{S1}}, \sigma_{\texttt{S1}} \rangle$ in the same instance.
- **Reception assertion.** If a non-faulty replica delivers $\langle d, \texttt{T}_{\texttt{F}}, \sigma_{\texttt{F}} \rangle$ in an EPBC instance, all non-faulty replicas must have received d in the same instance.

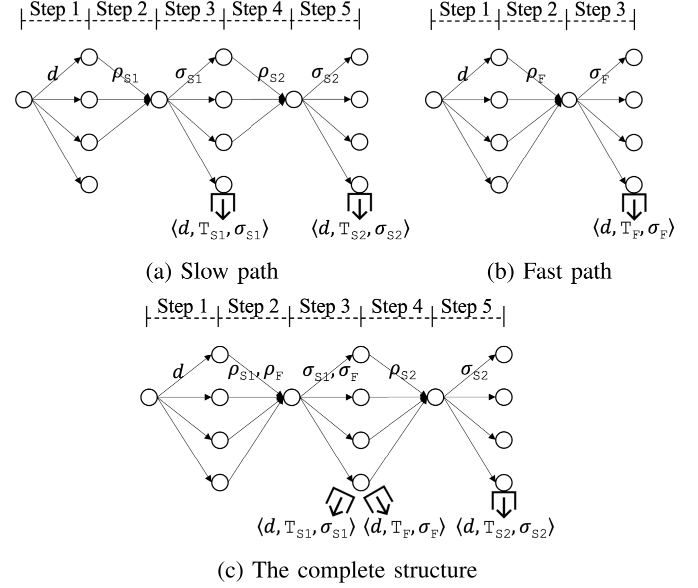


Fig. 2. The algorithm construction of EPBC ($\texttt{ALG}_{epbc}$).

2) Construction ($\texttt{ALG}_{epbc}$): We introduce $\texttt{ALG}_{epbc}$, a dual-path construction for EPBC comprising a slow path and a fast path, depicted in Fig. 2a and Fig. 2b, respectively. The slow path unfolds five communication steps. Initially, the broadcaster transmits data d to all replicas. Upon receipt, each replica produces a (n-f)-threshold signature share for $\langle d, \texttt{T}_{\texttt{S1}} \rangle$, denoted as $\rho_{\texttt{S1}}$, and sends it back to the broadcaster in the second step. Once n-f signature shares $\rho_{\texttt{S1}}$ are collected, the broadcaster generates a complete (n-f)-threshold signature $\sigma_{\texttt{S1}}$ and broadcasts it in the third step. At the end of the third step, a replica will deliver $\langle d, \texttt{T}_{\texttt{S1}}, \sigma_{\texttt{S1}} \rangle$ after receiving $\sigma_{\texttt{S1}}$. Also, it will generate another (n-f)-threshold signature share for $\langle d, \texttt{T}_{\texttt{S2}} \rangle$, labeled as $\rho_{\texttt{S2}}$, which is then sent to the broadcaster in the fourth step. Similarly, upon gathering n-f $\rho_{\texttt{S2}}$ shares, the broadcaster constructs a complete threshold signature $\sigma_{\texttt{S2}}$ and broadcasts it in the fifth step. Receiving $\sigma_{\texttt{S2}}$ prompts a replica to deliver $\langle d, \texttt{T}_{\texttt{S2}}, \sigma_{\texttt{S2}} \rangle$.

In contrast, the fast path involves three communication steps beginning similarly with the broadcaster dispatching data d. Each replica produces a n-threshold signature share on $\langle d, \texttt{T}_{\texttt{F}} \rangle$, denoted as $\rho_{\texttt{F}}$, and forwards this to the broadcaster in the second step. Upon collecting $\rho_{\texttt{F}}$ from all replicas, the broadcaster assembles a complete threshold signature, $\sigma_{\texttt{F}}$, and broadcasts it in the third step. Receiving $\langle d, \texttt{T}_{\texttt{F}}, \sigma_{\texttt{F}} \rangle$ enables a replica to deliver $\sigma_{\texttt{F}}$.

$\texttt{ALG}_{epbc}$ melds these two paths as shown in Fig. 2c. Replicas generate both $\rho_{\texttt{S1}}$ and $\rho_{\texttt{F}}$ by the end of the first step, which are then forwarded to the broadcaster in the second step. After obtaining n-f $\rho_{\texttt{S1}}$ shares, the broadcaster forms $\sigma_{\texttt{S1}}$, which is then broadcast in the third step. It continues to collect more messages during the second step. Receiving n $\rho_{\texttt{F}}$ shares prompts the broadcaster to further construct $\sigma_{\texttt{F}}$ for

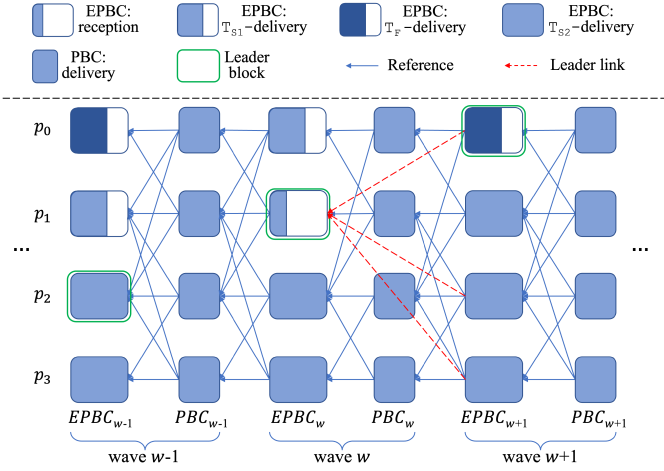


Fig. 3. The overall structure of Wahoo.

broadcasting. Thus, a replica will deliver $\langle d, \mathtt{T}_{\mathtt{S1}}, \sigma_{\mathtt{S1}} \rangle$ or/and $\langle d, \mathtt{T}_{\mathtt{F}}, \sigma_{\mathtt{F}} \rangle$ if it receives $\sigma_{\mathtt{S1}}$ or/and $\sigma_{\mathtt{F}}$ at the end of the third step. The process for the remaining two steps mirrors that of the slow path.

The validation predicates Q and P are set to the (n-f)-threshold signature verification function and n-threshold signature verification function, respectively. For brevity, we denote the deliveries with tags $\mathtt{T}_{\mathtt{S1}}$, $\mathtt{T}_{\mathtt{S2}}$, and $\mathtt{T}_{\mathtt{F}}$ as $\mathtt{T}_{\mathtt{S1}}$-delivery, $\mathtt{T}_{\mathtt{S2}}$-delivery, and $\mathtt{T}_{\mathtt{F}}$-delivery, respectively. $\mathtt{ALG}_{spbc}$ similarly incurs linear communication overhead. Furthermore, $\mathtt{T}_{\mathtt{S1}}$-delivery, $\mathtt{T}_{\mathtt{S2}}$-delivery, and $\mathtt{T}_{\mathtt{F}}$-delivery have latencies of 3δ, 5δ, and 3δ, respectively.

IV. WAHOODESIGN

We answer the question raised in Section II-B affirmatively, by proposing a novel DAG-based protocol named Wahoo. Built upon EPBC and PBC, Wahoo achieves quadratic communication overhead. Besides, EPBC's $\mathtt{T}_{\mathtt{F}}$-delivery mechanism allows Wahoo to deliver a low latency of 4δ under favorable conditions, matching GradedDAG's efficiency.

A. Overview of Wahoo

Generally speaking, Wahoo is designed by leveraging EPBC and PBC to broadcast blocks. As shown in Fig. 3, Wahoo operates through successive waves, where each wave comprises two distinct phases: the EPBC phase and the PBC phase. The EPBC phase involves block dissemination using EPBC, while the PBC phase utilizes PBC for block broadcasting. Blocks transmitted in these phases are termed EPBC blocks and PBC blocks, whose data structures are described in Algorithm 1. Both EPBC and PBC blocks include fields of payload (p), replica number (p), wave number (w), and hashes ($refs$) of blocks in the previous phase. An EPBC block will also include a partial threshold signature (sh) to implement the GPC function, while an EPBC block will include the leader link (ll) and proof (pf) to implement the leader-linking mechanism. Waves are sequentially numbered, each assigned an incrementally higher number w. Accordingly, the EPBC phase and the PBC phase in wave w are denoted as $EPBC_w$ and PBC_w, respectively.

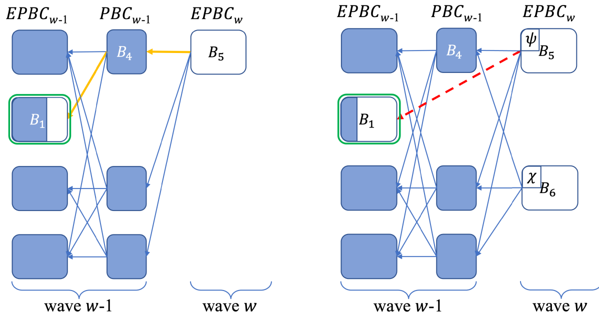
Algorithm 1 Data Structures & Utilities

1: **struct** $PbcBlock$\{
2: pl, p, w // payload, replica number, and wave number
3: $refs$ // hashes and tags of blocks in previous phase
4: sh // partial threshold signature to implement GPC
5: \}
6: **struct** $EPbcBlock$\{
7: pl, p, w // payload, replica number, and wave number
8: $refs$ // hashes of blocks in previous phase
9: ll, pf // leader link and proof
10: \}
11: **func** $GenPbcBlock$():
12: initialize a $PbcBlock$ as B
13: $B.pl \leftarrow$ outstanding txs from the replica's buffer
14: **return** B
15: **func** $GenEPbcBlock$():
16: initialize an $EPbcBlock$ as B
17: $B.pl \leftarrow$ outstanding txs from the replica's buffer
18: **return** B

A block in Wahoo must include at least n-f hashes of blocks from the previous phase, referred to as references. We say a block B directly references another block C if B contains C's hash. Additionally, if block C references block D, then block B is considered to indirectly reference D. Particularly, we stipulate that each block must directly reference its broadcaster's block in the previous phase. The blocks directly referenced by a block B are termed B's parents, while those it references, either directly or indirectly, are termed B's ancestors. Notably, a block is also an ancestor of itself.

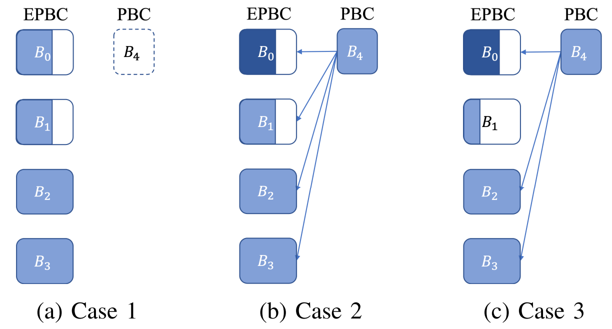
In an EPBC phase, if a replica successfully delivers a minimum of n-f blocks tagged as $\mathtt{T}_{\mathtt{S2}}$ or $\mathtt{T}_{\mathtt{F}}$, it can progress to the subsequent PBC phase. During the PBC phase, the replica creates a block referencing all blocks delivered in the previous EPBC phase, regardless of their tags. One replica is elected as the wave's leader via the GPC function. The EPBC block proposed by this leader is considered the wave's leader block. If a replica delivers the leader block tagged as $\mathtt{T}_{\mathtt{S2}}$ or $\mathtt{T}_{\mathtt{F}}$, as illustrated in waves $w-1$ and $w+1$ in Fig. 3, it can commit the leader block and its ancestor blocks. Upon delivering at least n-f PBC blocks, a replica can progress to the next wave by broadcasting a new EPBC block.

When an EPBC block cannot reference the previous wave's leader block, it must incorporate a leader-linking mechanism. This mechanism necessitates the inclusion of a hash value, termed the 'leader link', along with a corresponding proof, as shown by Line 9 in Algorithm 1. The mechanism has two cases: if the leader link is non-empty, the proof is referred to as an 'exclusive-commit proof'; otherwise, it is identified as a 'no-commit proof'. In essence, a non-empty leader link of a block L, along with the exclusive-commit proof, signifies that L is the exclusive leader block eligible for committing in the previous wave. Conversely, the combination of an empty leader link and a no-commit proof certifies that no leader



(a) Prev. leader is referenced (b) Prev. leader is not referenced

Fig. 4. Schematic diagram for the EPBC block construction.



(a) Case 1 (b) Case 2 (c) Case 3

Fig. 5. Schematic diagram for PBC block construction. Case 1 illustrates a scenario where fewer than n-f EPBC blocks are delivered with tags $\mathtt{T_{S2}}$ or $\mathtt{T_F}$, leading to the inability to construct B_4. In Case 2, two EPBC blocks are delivered with $\mathtt{T_{S2}}$, and one EPBC block is delivered with $\mathtt{T_F}$, allowing the construction of B_4. Case 3 exemplifies a situation where B_1 is received but not delivered, preventing its referencing in B_4.

block from the previous wave qualifies for committing. It is important to note that when determining the ancestor blocks of a given block, blocks interconnected through the leader links will also be counted. This means if an EPBC block B embeds L's hash as the leader link, the ancestors of L are also considered to be B's ancestors.

Algorithm 2 The EPBC Phase in Wave w (for Replica p_i)

1: **Let** S_b denote blocks delivered in phase PBC_{w-1}. Besides, let B represent the block to be broadcast by p_i in phase $EPBC_w$, and $hash$ refers to the hash function.
2: **Let** L denote the leader block in wave $w-1$.
3: **Let** $SignShr$ and $Comb$ denote threshold sig. functions.
4: $h \leftarrow hash(L)$, $\rho \leftarrow SignShr(h)$
5: **broadcast** $\langle \mathtt{RCEP}, h, \rho \rangle$

6: **upon** receiving n-f RCEP messages:
7: $h, c \leftarrow$ most frequently occurring hash and its frequency
8: **if** $c \geq f+1$ **then:**
9: $ll \leftarrow h$, $shares \leftarrow f$+1 threshold shares on h
10: $proof \leftarrow Comb(shares, h)$
11: **else:**
12: $ll \leftarrow \perp$
13: $proof \leftarrow$ all n-f RCEP messages

14: $\mathrm{S}_h \leftarrow \{h(b) \mid b \in \mathrm{S}_b\}$
15: $B \leftarrow GenEPbcBlock()$
16: $B.p \leftarrow i$, $B.w \leftarrow w$, $B.refs \leftarrow \mathrm{S}_h$
17: **if** L is ancestor of B **then:**
18: $B.ll \leftarrow \perp$, $B.pf \leftarrow \perp$
19: **else:**
20: $B.ll \leftarrow ll$, $B.pf \leftarrow proof$
21: **broadcast** B through the EPBC protocol

22: **upon** delivering n-f $EPBC_w$ blocks with tag $\mathtt{T_{S2}}$ or $\mathtt{T_F}$:
23: **advance** to the phase PBC_w

B. EPBC Phase

Each wave starts with an EPBC phase. Except for the first wave, each EPBC block must reference at least n-f blocks in the previous PBC phase. The EPBC phase is described in Algorithm 2. If the new EPBC block can indirectly reference the leader block of the previous wave, no additional actions are required (Lines 17-18 of Algorithm 2). For instance, in Fig. 4a, the EPBC block B_5 indirectly references the previous leader block B_1, eliminating the need for B_5 to include extra fields. On the contrary, if the EPBC block cannot reference the previous leader block, it necessitates a leader link accompanied by a proof. As depicted in Fig. 4b, the EPBC block B_5 contains a leader link that points to B_1 along with an exclusive-commit proof ψ, whereas the EPBC block B_6 includes an empty leader link plus a no-commit proof χ.

To generate the leader link and the corresponding proof, replicas exchange 'reception' information about the previous leader block by broadcasting messages in the format $\langle \mathtt{RECP}, h, \rho \rangle$ (Lines 4-5). Here, h represents the hash of the previous leader block, and ρ is a (f+1)-threshold signature share on h. Upon receiving a $\langle \mathtt{RECP}, h, \rho \rangle$ message, a replica that lacks the corresponding block will use the block retrieval mechanism (Section IV-D) to acquire the absent block data from the message's sender.

Upon receiving n-f RECP messages, a replica identifies the most frequently occurring hash h, with its frequency noted as c. If c equals or exceeds f+1, the replica will generate a complete (f+1)-threshold signature on h using the signature shares contained in RECP messages. The EPBC block includes h as the leader link and the complete signature as the exclusive-commit proof. Otherwise, if c is less than f+1, the leader link is set as empty, and all received RECP messages are packaged into the EPBC block, serving as the no-commit proof (Lines 6-20 of Algorithm 2). The constructed EPBC block will then be broadcast through EPBC.

Upon receiving an EPBC block B, a replica will first check whether it has delivered all of B's parent blocks. If any are missing, the replica proceeds to retrieve the absent data from B's constructor, as elaborated in Section IV-D. Only if all of B's parent blocks have been delivered can the replica participate in the EPBC process of B.

C. PBC Phase

A replica advances to the PBC phase only after delivering at least n-f blocks in the previous EPBC phase with tags $\mathtt{T_{S2}}$ or $\mathtt{T_F}$, as described in Lines 22-23 of Algorithm 2. Taking

Algorithm 3 The PBC Phase in Wave w (for Replica p_i)

1: **Let** S_b denote blocks delivered in phase $EPBC_w$, while T_b signifies all blocks being broadcast in $EPBC_w$. Furthermore, let B represent the block to be broadcast by p_i in phase PBC_w.
2: $\mathrm{S}_h \leftarrow \{\langle hash(b), b.tag\rangle \mid b \in \mathrm{S}_b\}$
3: $B \leftarrow GenPbcBlock()$
4: $B.p \leftarrow i$, $B.w \leftarrow w$, $B.refs \leftarrow \mathrm{S}_h$
5: $B.sh \leftarrow SignShr(w)$
6: **broadcast** B through the PBC protocol

7: **for each** b in T_b **do**
8:    **if** $b \notin \mathrm{S}_b$ **then:**
9:       terminate its participation in b's EPBC process

10: **upon** receiving f+1 PBC blocks in phase PBC_w:
11:    $shares \leftarrow$ all threshold shares in these blocks
12:    $sig \leftarrow Comb(shares, w)$
13:    **parse** sig as an integer number g
14:    $l \leftarrow g\%n$
15:    $L_w \leftarrow b$**if**$b \in \mathrm{S}_b$ and $b.i = l$
16:    **if** L_w is delivered with the tag $\mathtt{T_{S2}}$ or $\mathtt{T_F}$ **then:**
17:       **commit** L_w and its ancestors
18: **upon** delivering n-f PBC_w blocks:
19:    **advance** to the phase $EPBC_{w+1}$

Fig. 5a as an example, where $n = 4$ replicas are involved and $f = 1$, the replica cannot create a PBC block B_4 since it only delivers two blocks B_2 and B_3 with the tag $\mathtt{T_{S2}}$, which is less than n-f. However, this case represents a transitional state, and it is expected that additional EPBC blocks with tags $\mathtt{T_{S2}}$ or $\mathtt{T_F}$ will soon be delivered. As illustrated in Fig. 5b, after delivering one block with tag $\mathtt{T_F}$ (i.e., B_0), the replica can construct the PBC block B_4. A PBC block can reference all delivered blocks in the previous EPBC phase, irrespective of their tags. Fig. 5c shows block B_4 referencing blocks B_0, B_2, and B_3, and even B_1, which is delivered with tag $\mathtt{T_{S1}}$. On the contrary, if the EPBC block is only received, exemplified by B_1 in Fig. 5c, it will not be referenced by the new PBC block. A PBC block will specify delivery tags of its parent blocks in the block using a map structure, with the parent block hash as the key and the delivery tag as the value.

The PBC phase is outlined in Algorithm 3. Once a replica constructs the PBC block B, it will stop its involvement in the EPBC process of a block b if b is not referenced by B (Lines 7-9). Similar to the EPBC phase, a replica participates in the broadcast process of a PBC block B only if it has received or delivered all blocks referenced by B. It also makes use of the block retrieval mechanism to acquire its missing blocks.

An (f+1)-threshold signature share on the wave number w is generated and included in the PBC block. This signature share serves to implement the GPC function, configured with the threshold parameter in GPC set to f+1. When a replica receives f+1 valid PBC blocks, it can generate a complete threshold signature, subsequently converted into an integer denoted as g. The output of the GPC function, representing the leader's replica number, is established as g modulo n. The EPBC block proposed by the leader is named the leader block for this wave, labeled as L_w (Lines 10-15). Should L_w be delivered with the tag $\mathtt{T_{S2}}$ or $\mathtt{T_F}$, it can be committed directly (Lines 16-17). At the same time, the ancestor blocks of L_w can also be committed, as detailed in Section IV-E.

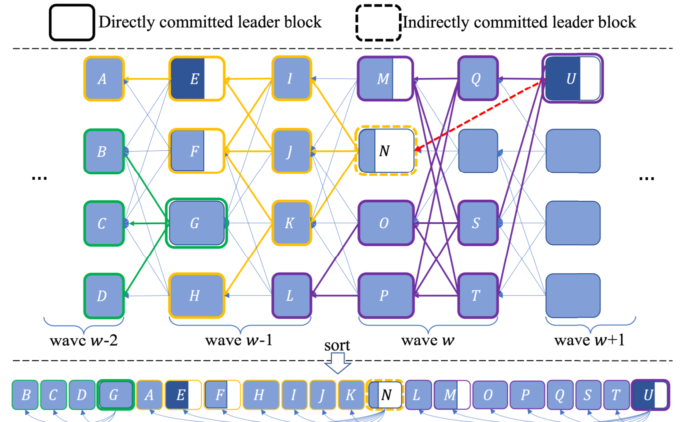


Fig. 6. An example to show the process of committing blocks.

D. Block Retrieval Mechanism

The block retrieval mechanism in Wahoo is implemented through an interactive mechanism between the requester and the responder. The responder is the constructor of a block B, while the requester is a replica who receives B but has not received or delivered all ancestors of B.

The requester sends a request message $\langle \mathtt{REQ}, h, t\rangle$ to the responder, where h is the hash of the missing block C, and t is a tag. When C is PBC block, t is set to $\mathtt{PBC}$. For EPBC blocks linked through references, t is an element in $\{\mathtt{T_{S1}}, \mathtt{T_{S2}}, \mathtt{T_F}\}$. If linked via the leader link, t is set to $\mathtt{RECP}$. Besides, multiple blocks can be retrieved in one request by including multiple pairs of h and t, thus increasing the retrieval efficiency.

Upon receiving the request message, the responder addresses each pair of h and t in the message. If t is in $\{\mathtt{PBC}, \mathtt{T_{S1}}, \mathtt{T_{S2}}, \mathtt{T_F}\}$, the response includes the block data and a corresponding certificate—σ for $\mathtt{PBC}$, $\sigma_{\mathtt{S1}}$, $\sigma_{\mathtt{S2}}$, or $\sigma_{\mathtt{F}}$ for $\mathtt{T_{S1}}$, $\mathtt{T_{S2}}$, or $\mathtt{T_F}$. For $\mathtt{RECP}$, the response only includes block data. After receiving the response, the requester validates the block, particularly the correctness of the included certificate, before accepting or delivering it.

The above request-and-response process is recursive. If blocks referenced by the newly received block are missing, the requester initiates another retrieval request. This recurs until all of B's ancestor blocks are received or delivered.

E. Block Committing Mechanism

In Wahoo, the process of committing blocks depends on leader blocks, categorized as direct or indirect. Direct committing occurs when a leader block, denoted as B, is delivered with tag $\mathtt{T_{S2}}$ or $\mathtt{T_F}$. Following this event, a replica thoroughly tracks down all ancestor blocks of B to identify any uncommitted leader blocks. If an uncommitted leader

Algorithm 4 The Mechanism to Commit Blocks

1: **Let** $\mathcal{A}_B$ represent the ancestors of a block B and $\mathcal{S}$ represent the already committed block sequence.
2: **Let** L_c denote the leader block to be committed.
3: $\mathcal{U} \leftarrow$ uncommitted leader blocks in $\mathcal{A}_{L_c}$
4: **sort** $\mathcal{U}$ based on the wave number
5: **for** each u in $\mathcal{U}$ **do:**
6: $\quad \mathcal{H} \leftarrow \mathcal{A}_u \setminus \mathcal{S}$
7: $\quad$ **sort** $\mathcal{H}$ first by wave number, within the same wave prioritize EPBC over PBC, and then sort by broadcaster's replica number within the same phase.
8: $\quad \mathcal{S} \leftarrow \mathcal{S} + \mathcal{H}$

block, such as C, is detected, the process to commit it is prompted by the committing event of B, referred to as the indirect committing of C. For instance, as illustrated in Fig. 6, leader blocks G and U demonstrate direct committing, while N showcases indirect committing, triggered by the committing of U. These committing actions occur sequentially, aligning with the ascending order of their assigned wave numbers.

In a DAG-based system like Wahoo, committing blocks involves sorting them to form a block sequence. When a leader block is committed, either directly or indirectly, it initiates by eliminating all already committed blocks from its set of ancestor blocks. The remaining blocks undergo sorting first according to their wave numbers. Within each wave, blocks from the EPBC phase precede those from the PBC phase. Further, within the same phase, blocks are sorted by the broadcaster's replica number. This sorted collection of blocks is then appended to the existing sequence of committed blocks. For example, in Fig. 6, committing the leader block G leads to the sorting of its ancestor blocks into the sequence B, C, D, and G. Committing the leader block U triggers the committing of the leader block N. N then sorts its ancestors into the sequence A, E, F, H, I, J, K, and N, appended after G. Further, U sorts its ancestors into the sequence L, M, O, P, Q, S, T, and U, appended after N. The pseudocode outlining the process of block committing is described in Algorithm 4.

F. Performance Analysis

With $\texttt{ALG}_{pbc}$ and $\texttt{ALG}_{spbc}$ boasting linear communication overhead, Wahoo significantly reduces its communication load to $O(n^2)$. This stands in stark contrast to the $O(n^3)$ overhead prevalent in existing DAG-based BFT protocols. Under optimal conditions, all instances of EPBC within Wahoo successfully complete with a $\texttt{T}_\texttt{F}$ tag after just three communication steps and transition to the subsequent PBC phase. In these ideal scenarios, the GPC output during the first communication step of the PBC phase efficiently reveals the leader block in the wave. Consequently, the time taken for a leader block to be committed is a mere 4δ, aligning with the best performance of leading DAG-based BFT protocols such as GradedDAG [45].

However, in less favorable scenarios, an EPBC instance may require 5δ to produce an output, extending the completion time for an entire wave to 8δ. Given the approximate $2/3$ probability of committing a leader block, the expected number of waves required for this averages around 1.5. Consequently, in these suboptimal conditions, the anticipated latency for committing a leader block could extend to 12δ.

V. Correctness Analysis of Wahoo

The correctness of Wahoo includes three aspects: safety, liveness, and completeness, which are analyzed respectively in this section.

A. Safety Analysis

In accordance with Section IV-E, committed blocks collectively form a block sequence, denoted as $\mathcal{S}$. Each committed block within this sequence is uniquely identified by a distinct sequence number represented as $\langle i, B\rangle$, where i signifies its unique sequence identifier. Moreover, since this block sequence is generated based on leader blocks, these leaders collectively form their own sequence, termed 'leader sequence' and denoted as $\mathcal{L}$. A relationship is established between these sequences using the prefix relationship, symbolized as $\prec$ and defined as follows: if $\mathcal{S}_1$ equals the initial segment of $\mathcal{S}_2$ (expressed as $\mathcal{S}_2[0\colon \mathcal{S}_1.\text{len}]$), then $\mathcal{S}_1$ is considered a prefix of $\mathcal{S}_2$, denoted by $\mathcal{S}_1 \prec \mathcal{S}_2$. The same logic applies to leader sequences: if $\mathcal{L}_1$ equals $\mathcal{L}_2[0\colon \mathcal{L}_1.\text{len}]$, then $\mathcal{L}_1$ is a prefix of $\mathcal{L}_2$, symbolized as $\mathcal{L}_1 \prec \mathcal{L}_2$. The relationship inherently upholds the reflexive property, signifying that a sequence always serves as a prefix to itself.

The safety property of Wahoo is expressed in Theorem 3, whose proof relies on Lemma 1 and Lemma 2.

Lemma 1: If two leader blocks L_1 and L_2 are committed directly, then either L_1 is an ancestor of L_2 or L_2 is an ancestor of L_1.

Proof: If L_1 and L_2 are from the same wave, it implies they are delivered through the same EPBC instance. As per the consistency property of EPBC, in such a case, L_1 must be equal to L_2, establishing that L_1 is an ancestor of L_2. Otherwise, without loss of generality, we assume the wave number of L_1, denoted as w_1, is smaller than that of L_2, denoted as w_2. Since L_1 is committed directly, it must be delivered with tag $\texttt{T}_\texttt{S2}$ or $\texttt{T}_\texttt{F}$.

If L_1 is delivered with $\texttt{T}_\texttt{S2}$, based on the delivery assertion property of EPBC, at least $n-2f$ non-faulty replicas must have delivered L_1 with $\texttt{T}_\texttt{S1}$ and will reference L_1 in the subsequent PBC phase. Given that $n-2f \geq f+1$ and a block in the subsequent EPBC phase references at least n-f blocks in the PBC phase, each block in a wave w ($w \geq w_1$) will indirectly reference L_1. Hence, L_2 must indirectly reference L_1.

If L_1 is delivered with $\texttt{T}_\texttt{F}$, following the reception assertion property of EPBC, all non-faulty replicas must have received L_1 and will broadcast a $\texttt{RECP}$ message containing the hash of L_1 before advancing to the next wave w_1+1. Consequently, if an EPBC block in wave w_1+1 cannot reference L_1 indirectly, it will receive at least $n-2f$ matching $\texttt{RECP}$ messages and can only generate a leader link at L_1. Thus, each block in a wave w ($w \geq w_1$), including L_2, will also reference L_1.

In conclusion, L_1 must be an ancestor, thus establishing the validity of the lemma. □

Lemma 2: If two non-faulty replicas p_i and p_j commit two block sequences $\mathcal{S}_i$ and $\mathcal{S}_j$, then either $\mathcal{S}_i \prec \mathcal{S}_j$ or $\mathcal{S}_j \prec \mathcal{S}_i$.

Proof: Suppose $\mathcal{S}_i$ and $\mathcal{S}_j$ are committed based on two directly committed leader blocks, L_i and L_j, respectively. Lemma 1 establishes that either L_i is an ancestor of L_j or vice versa. Without loss of generality, assume L_i is an ancestor of L_j. When p_j commits L_j, it traces back through all leader blocks in its ancestors, which includes L_i. If p_j has already committed L_i, then the block sequence $\mathcal{S}_i$ produced by L_i, is pre-existing in p_j. If p_j has not yet committed L_i, it will first sort all ancestor blocks of L_i into a block sequence, precisely identical to $\mathcal{S}_i$. Subsequently, p_j removes blocks in $\mathcal{S}_i$ from its ancestors and sorts the remaining blocks. These sorted blocks are appended to $\mathcal{S}_i$, forming $\mathcal{S}_j$. Thus, $\mathcal{S}_i \prec \mathcal{S}_j$. □

Theorem 3 (Safety): If two non-faulty replicas p_i and p_j commit two blocks B_i and B_j with the same sequence number, B_i must be equal to B_j.

Proof: Let B_i and B_j be included in two block sequences $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively, with their sequence number being m. According to Lemma 2, either $\mathcal{S}_i \prec \mathcal{S}_j$ or vice versa. Without loss of generality, assume $\mathcal{S}_i \prec \mathcal{S}_j$, namely $\mathcal{S}_i = \mathcal{S}_j[0\colon \mathcal{S}_i.\text{len}]$. Therefore, $\mathcal{S}_i[m] = \mathcal{S}_j[0\colon \mathcal{S}_i.\text{len}][m] = \mathcal{S}_j[m]$. On the other hand, since $B_i = \mathcal{S}_i[m]$ and $B_j = \mathcal{S}_j[m] = B_j$, it follows that $B_i = B_j$, confirming the safety of Wahoo. □

B. Liveness Analysis

The liveness property is articulated through Theorem 5, firmly established upon the foundation laid by Lemma 4.

Lemma 4: Starting from any given moment t_0, the probability of directly committing a leader block approaches 1 as time progresses.

Proof: Consider the moment when the leader of a wave is revealed through the GPC function. The unpredictability property of GPC establishes that at least one non-faulty replica has broadcast the PBC block of this wave. This non-faulty replica must have delivered at least n-f EPBC blocks of this wave with the tag $\texttt{T}_{\texttt{S2}}$ or $\texttt{T}_{\texttt{F}}$. Once the leader block happens to be one of these EPBC blocks, the likelihood of directly committing the leader block is at least $\frac{n-f}{n}$. Given $n \geq 3f+1$, this likelihood is greater than $2/3$. In other words, the probability of the leader block being directly committed in a wave denoted as p, exceeds $2/3$. Consequently, the probability of at least one leader block being directly committed within k waves after t_0 is $1-(1-p)^k$, and this probability approaches 1 as k increases. □

Theorem 5 Liveness: If a client submits a transaction tx to Wahoo, tx will eventually be committed.

Proof: Let t_0 denote the time when tx is received by all non-faulty replicas. We define a block B to directly include a request tx if tx is contained in B. Additionally, we say that B indirectly includes tx if tx is contained in a block C and C is referenced by B. Upon committing B, all requests included by B, whether directly or indirectly, are committed as well.

In the subsequent phase after t_0, denoted as s, all blocks proposed by non-faulty replicas will include tx directly or indirectly. As a block in the following phase must reference at least n-f blocks in the previous phase, each block in a phase after s, including the leader block, will definitely include tx. Let t_1 be the time when the first replica advances to the phase following s. According to Lemma 4, the probability of a leader block being directly committed after t_1 approaches 1, consequently committing tx as well. Therefore, tx will eventually be committed. □

C. Completeness Analysis

Theorem 6 Completeness: For each sequence number k, each non-faulty replica will commit a block numbered k eventually.

Proof: Since a block references at least n-f blocks from the preceding phase, directly committing a leader block will result in committing at least $n-f+1$ blocks. Therefore, as long as a replica can keep directly committing leader blocks, the number of its committed blocks will continue to rise, allowing it to commit a block for each sequence number k. Consequently, in the following parts, we focus on demonstrating that each non-faulty replica can continuously commit leader blocks directly.

According to the proof of Lemma 4, when a wave's leader is revealed, a non-faulty replica, denoted as p_i, must have delivered at least n-f EPBC blocks of this wave, tagged either $\texttt{T}_{\texttt{S2}}$ or $\texttt{T}_{\texttt{F}}$. If p_i can directly commit the wave's leader block, denoted as L, it must have referenced L with tag $\texttt{T}_{\texttt{S2}}$ or $\texttt{T}_{\texttt{F}}$ in its next PBC block, which we refer to as B. A non-faulty replica that receives B will check whether B's parent blocks have been delivered and retrieve the missing ones. As a result, each non-faulty replica will eventually deliver L with tag $\texttt{T}_{\texttt{S2}}$ or $\texttt{T}_{\texttt{F}}$ and directly commit L. In other words, with a probability of exceeding $2/3$, each non-faulty replica is capable of directly committing the leader block in every wave. Thus, each non-faulty replica can continuously commit leader blocks directly, and the theorem is established. □

VI. Implementation & Evaluation

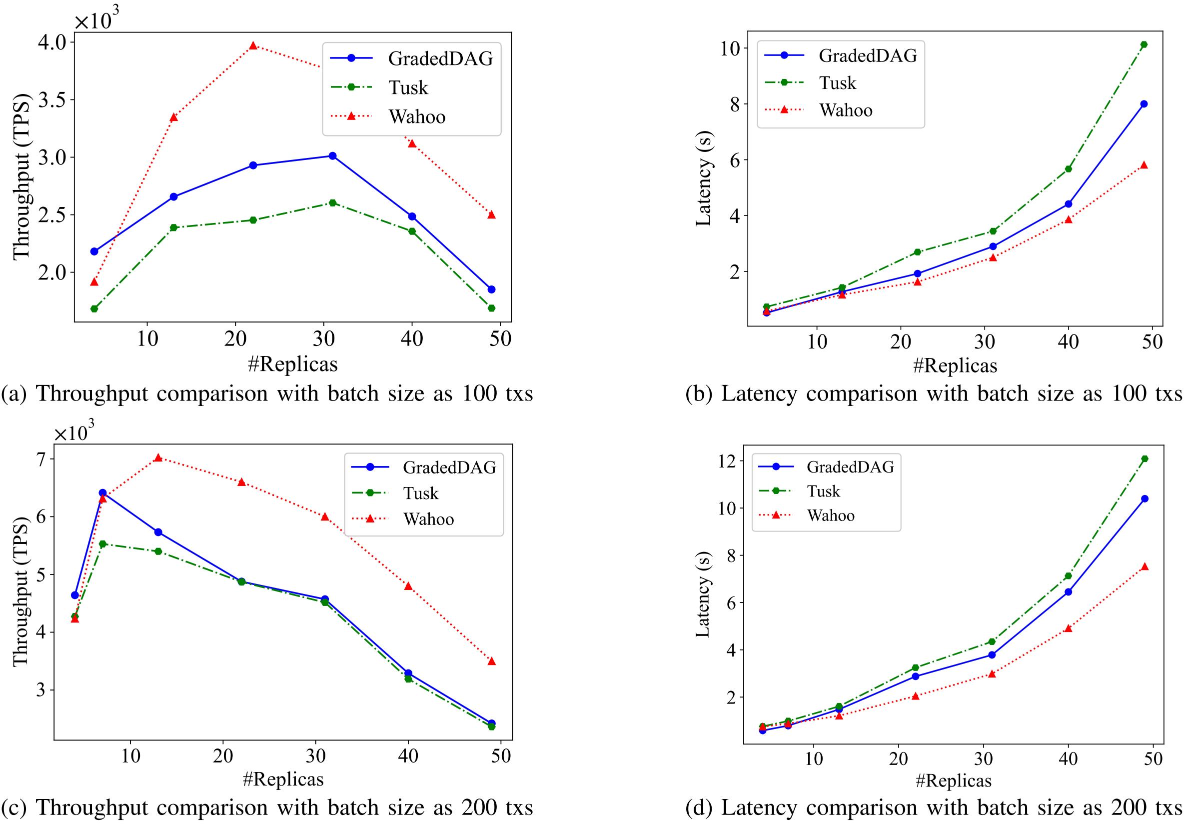
In this section, we present a comprehensive evaluation of Wahoo's effectiveness, comparing it against state-of-the-art counterparts—Tusk and GradedDAG.

A. Implementation & Setting

To ensure fairness in our assessment, we implement Tusk, GradedDAG, and Wahoo using the same framework, which is written in Golang and comprises approximately 4,100 lines of code. We are open-sourcing our implementation.[1] Our implementation leverages reputable open-source libraries, such as dedis/kyber[2] for the threshold signature and hashicorp/go-msgpack[3] for the communication between replicas. The transaction size is consistently set at 250 bytes.

Our performance evaluation centers around two key metrics: latency and throughput. Latency is measured as the average

[1] https://github.com/CGCL-codes/Wahoo
[2] https://github.com/dedis/kyber
[3] https://github.com/hashicorp/go-msgpack



(a) Throughput comparison with batch size as 100 txs
(b) Latency comparison with batch size as 100 txs
(c) Throughput comparison with batch size as 200 txs
(d) Latency comparison with batch size as 200 txs

Fig. 7. Performance comparison under good situations.

time taken to commit a transaction from the moment it is proposed by the client. Throughput is computed as the average number of transactions committed per second, commonly abbreviated as *Transactions Per Second* (TPS). We assess both situations—good situations without any faulty replicas and suboptimal situations with some faulty replicas.

Experiments are conducted on Alibaba Cloud, with replicas distributed across six regions in a geo-distributed manner: Japan (Tokyo), South Korea (Seoul), Singapore, the US (Silicon Valley), the UK (London), and Germany (Frankfurt). Specifically, when x replicas are involved in a group of experiments, at least $\lfloor x/6 \rfloor$ replicas and at most $\lceil x/6 \rceil$ are deployed in each region, ensuring that replicas are distributed as evenly as possible. Each replica is deployed as an ECS.g6e.xlarge instance, featuring 4 vCPUs, 16 GB memory, and a 40 GB SSD disk, running the Linux Ubuntu operating system. Each pair of replicas is interconnected through a network link with a bandwidth of 100 Mbps. To enhance the experimental accuracy, we repeat each group of experiments three times and take the averages to draw the graphs.

B. Performance Under Good Situations

In situations without any faulty replicas, we conduct a detailed performance evaluation among different protocols, considering the increasing number of replicas. Additionally, we examine the trade-off between throughput and latency, exploring the protocols' peak throughput capabilities.

1) Basic Performance Comparison: We conduct experiments in two sets, with batch sizes configured to 100 and 200 transactions, respectively. For each set, we scale the number of replicas from 4 to 49, and the results are depicted in Fig. 7. Across both sets, Wahoo outperforms its counterparts in terms of throughput and latency, with the exception of a few scenarios involving a smaller replica count. Notably, at a batch size of 200 transactions and with 49 replicas, Wahoo achieves 1.48x and 1.35x higher throughput compared to Tusk and GradedDAG. Regarding latency, Wahoo reduces it by 42.6% and 27.3%, respectively. The inferior performance observed in Wahoo when operating with fewer replicas can be attributed to the additional time required to collect all (i.e., 4) signature shares for data delivery tagged $\mathtt{T_F}$. This is in contrast to GradedDAG, which only requires two rounds for collecting n-f (i.e., 3) votes.

An intriguing phenomenon in Fig. 7 is that the throughput of all protocols experiences an initial increase followed by a decrease as the number of replicas increases, regardless of the batch size. This trend is due to the increased number of blocks proposed in each phase as the replica count rises from a small number, leading to more blocks being committed per second and an improved throughput. However, as the replica count continues to increase, the network becomes saturated, leading to congestion that negatively impacts throughput. Besides, a comparison between Fig. 7a and Fig. 7c reveals that with a larger batch size, the saturation point is reached sooner as the replica count increases, which aligns with expectations since larger batches consume more network resources.

2) Trade-off Between Throughput and Latency: In this set of experiments, we vary the number of replicas to 7 and

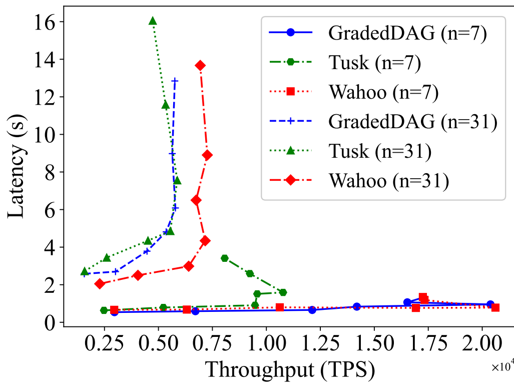


Fig. 8. Throughput v.s. latency under good situations.

31, gradually increasing the batch size to measure the peak throughput achieved by each protocol. The results are depicted in Fig. 8. With 7 replicas, both GradedDAG and Wahoo exhibit superior performance compared to Tusk, showcasing lower latency and higher peak throughput. In the scenario involving 31 replicas, Wahoo continues to surpass GradedDAG and Tusk. To be more specific, Wahoo achieves 1.26x and 1.23x higher throughput compared to GradedDAG and Tusk, respectively. In terms of latency reduction, Wahoo diminishes it by 20.6% and 24.6% compared to GradedDAG and Tusk, respectively. The enhanced performance of Wahoo is largely attributed to its reduced communication overhead.

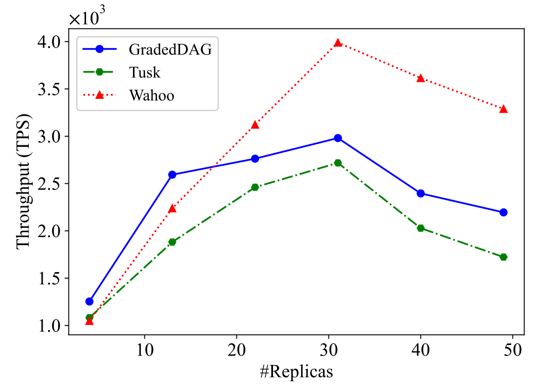
C. Performance With Faulty Replicas

Wahoo's optimal latency of 4δ is achieved under ideal scenarios, where all replicas are non-faulty, and the EPBC instance delivers data with tag $\mathtt{T_F}$. This ideal scenario has been thoroughly assessed in Section VI-B. On the other hand, it is also essential to assess Wahoo's performance in less favorable settings, where some replicas might be faulty. In this section, we simulate faulty replicas using simple crash faults.

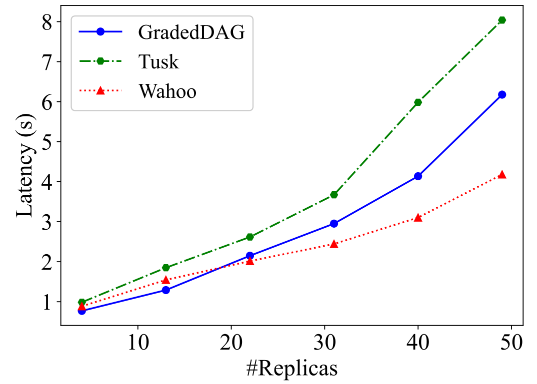
1) Basic Performance Comparison: We examine throughput and latency concerning the increasing number of replicas. Specifically, we set the number of faulty replicas, denoted as c, to half of the fault tolerance value f. For instance, if f=10, c would be set to $f/2$=5. The batch size remains set to 100 transactions. Experimental results, detailed in Fig. 9, illustrate that when the replica count reaches or exceeds 22, Wahoo outperforms other protocols due to its efficient communication overhead. Notably, in the scenario involving 49 replicas, Wahoo showcases a remarkable enhancement, elevating throughput by 90.9% (or 49.8%) and reducing latency by 48.0% (or 32.3%), compared to Tusk (or GradedDAG).

On the contrary, in scenarios with a small number of replicas, Wahoo might deliver lower throughput and higher latency than GradedDAG. This occurrence arises from Wahoo's inability to commit through $\mathtt{T_F}$-delivery due to some faulty replicas. Nevertheless, even when involving 14 replicas, Wahoo surpasses Tusk either in throughput or latency.

2) Trade-off Between Throughput and Latency: We also set the number of replicas to half of the fault tolerance value



(a) Throughput comparison with faulty replicas



(b) Latency comparison with faulty replicas

Fig. 9. Performance comparison with faulty replicas.

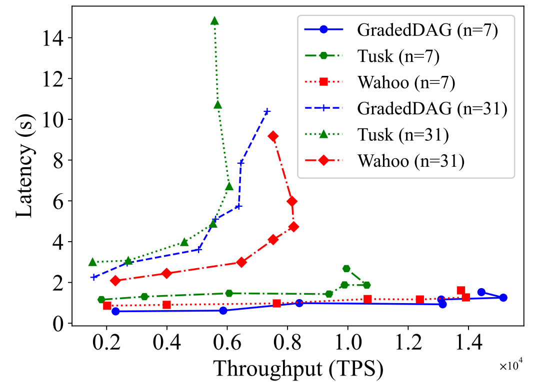
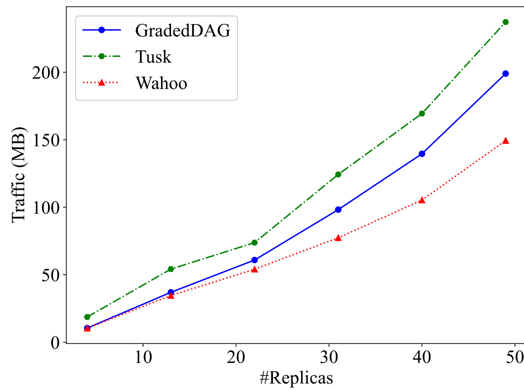
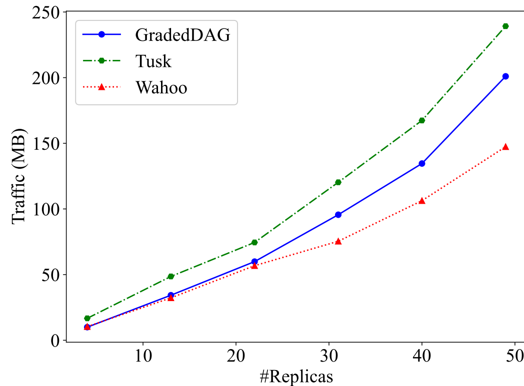


Fig. 10. Throughput v.s. latency with faulty replicas.

in this group of experiments. Specifically, for system scales of 7 and 31, we set c to 1 and 5, respectively. Fig. 10 depicts these experimental results. When the replica count is 7, Wahoo exhibits slightly lower peak throughput than GradedDAG. This disparity also stems from Wahoo's inability to commit through the $\mathtt{T_F}$-delivery, resulting in a higher consensus latency. Nevertheless, it still achieves a significantly higher peak throughput, approximately 1.31x greater than Tusk. In the scenario involving 31 replicas, Wahoo surpasses its counterparts in both throughput and latency, owing to its optimized communication overhead.



(a) Outbound traffic comparison



(b) Inbound traffic comparison

Fig. 11. Network traffic comparison.

D. Communication Overhead Comparison

To evaluate the effectiveness of Wahoo in reducing communication overhead, we conduct a group of experiments to compare the network traffic generated by different protocols. For this group of experiments, we maintain a consistent batch size of 100 transactions while varying the number of replicas from 4 to 49. All replicas are configured as non-faulty, and each protocol is operated for 50 waves in each configuration. We utilize the iftop[4] tool to measure the network traffic. To isolate the traffic resulting solely from the consensus protocols, we conduct the experiments in two steps. Firstly, we run only the operating system and measure the network traffic of a clean system, denoted as T_c. Secondly, we launch the consensus protocol and measure the total traffic, denoted as T_s. The traffic resulting from the consensus protocol is calculated as $T_s - T_c$. The measured results reveal that T_c is significantly smaller than T_s. We measure both inbound and outbound traffic, averaging the data across all replicas.

Experimental results are shown in Fig. 11. It is easy to find, regardless of whether we consider inbound or outbound traffic, Wahoo consistently incurs lower network traffic compared to its counterparts. Furthermore, as the number of replicas rises, the increase in Wahoo's network traffic is more gradual than that of the others. This disparity stems from the fact that Wahoo is designed with a low communication overhead of $O(n^2)$, in contrast to both GradedDAG and Tusk, which suffer from a higher communication overhead of $O(n^3)$.

[4]https://pdw.ex-parrot.com/iftop/

E. Evaluation of the Block Retrieval Mechanism

As a critical component in Wahoo, the block retrieval mechanism guarantees totality that if a non-faulty replica delivers a block, all other non-faulty replicas can do the same. Without the block retrieval mechanism, Wahoo must incorporate a simple 'block echo' mechanism to achieve totality. Specifically, the block echo mechanism requires that a replica broadcast any blocks it has delivered in this phase before moving to the next phase. We introduce a variant of Wahoo named Wahoo*, which substitutes the block retrieval mechanism with the block echo mechanism for comparison.

We evaluate the block retrieval mechanism by comparing Wahoo and Wahoo*, with a consistent setup of seven non-faulty replicas across all tests. We incrementally increase the batch size from 100 to 1,200 transactions, conducting each group of experiments for 50 waves. The results, depicted in Fig. 12a and Fig. 12b, demonstrate that Wahoo significantly outperforms Wahoo* in both throughput and latency metrics. Additionally, we compare the number of blocks transmitted under each mechanism, whose results are shown in Fig. 12c, which highlights the efficiency of the block retrieval mechanism over the block echo mechanism. Remarkably, at a batch size of 1,200 transactions, the number of blocks transmitted by Wahoo is only 1.51% of that by Wahoo*. In conclusion, the block retrieval mechanism effectively reduces the number of transmitted blocks, proving to be more communication-efficient than its counterpart (i.e., block echo mechanism).

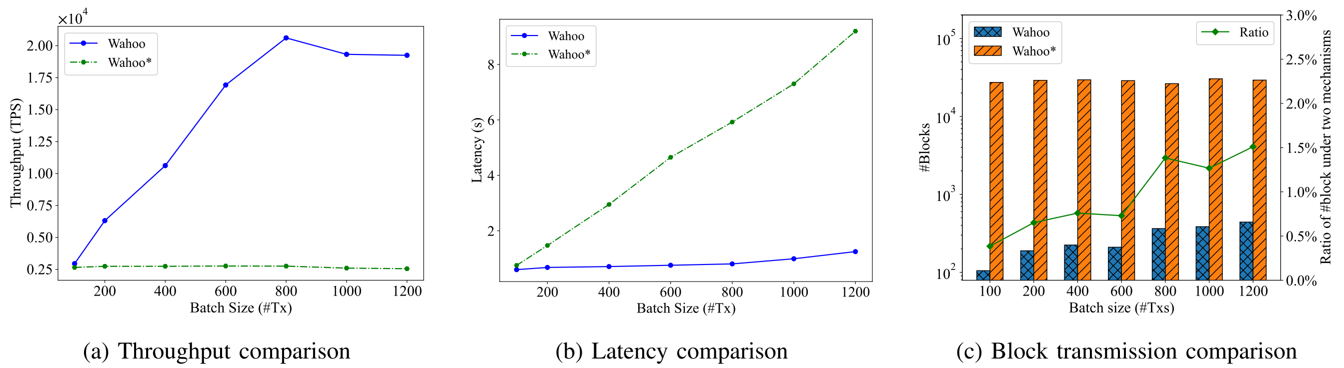
VII. Related Work

Given that Wahoo operates as an asynchronous protocol, our discussion primarily focuses on related work within the context of an asynchronous network. This includes both non-DAG asynchronous protocols and DAG-based asynchronous protocols. Additionally, we also talk about some interesting studies that, while not strictly classified as asynchronous protocols, bear significant relevance to Wahoo.

A. Non-DAG Asynchronous Protocols

Research on asynchronous BFT protocols has its origins in the 1980s, marked by pioneering *Asynchronous Binary Agreement* (ABA) protocols. As the simplest asynchronous consensus primitive, ABA facilitates agreement on binary values, whose representatives include Ben-Or [6], FMR [21], MMR [39], and ABY [1]. However, due to their binary limitation, ABA cannot directly serve as the consensus mechanism in blockchain systems.

To agree on arbitrary values, the concept of *Validated Asynchronous Byzantine Agreement* (VABA) or *Multivalued Validated Byzantine Agreement* (MVBA) is introduced. These VABA protocols are primarily divided into two groups based on their design principles: ABA-based and view-change-based. ABA-based VABA, exemplified by CKPS [11] and



(a) Throughput comparison (b) Latency comparison (c) Block transmission comparison

Fig. 12. Comparison between the block retrieval mechanism and the block echo mechanism.

Dumbo-MVBA [35], iterates ABA instances to agree on the broadcasted value. In contrast, view-change-based VABA, such as AMS [3] and sMVBA [22], utilizes a locking mechanism to restrict replicas to broadcast a consistent value across different views. While VABA applies to blockchain systems, they can only reach consensus on one block at a time, processing blocks sequentially, and limiting system throughput.

Another stream of works introduces a partially-synchronous path to asynchronous BFT protocol [7], [15], [23], [34], capable of delivering a high performance akin to partially-synchronous protocols under favorable conditions, while ensuring liveness comparable to asynchronous protocols in less ideal scenarios. However, these protocols also process data sequentially, constraining system throughput significantly.

B. DAG-Based Asynchronous Protocols

To parallel data processing, the DAG structure has been incorporated into the BFT consensus design, initially pioneered by DAGRider [28]. Successive efforts inspired by DAGRider strive to diminish consensus latency. For instance, Tusk reduces the wave length without compromising safety or liveness, achieving a good-case latency of 7 communication steps [16]. BullShark introduces an optimistic path that attains a good-case latency of 6 communication steps [40]. GradedDAG, utilizing GRBC and CBC protocols for block broadcasting, further reduces the wave length to two rounds, thus achieving a good-case latency of 4 steps [45]. Similarly, leveraging CBC protocols to broadcast blocks, LightDAG presents two variants to reduce good-case latencies, respectively [14]. Despite the advances, these protocols overlook the substantial communication overhead of $O(n^3)$. This is addressed by Wahoo, which reduces communication overhead to $O(n^2)$ while maintaining a low latency of 4 steps.

An additional notable contribution is Narwhal [16], which, rather than proposing a DAG-based consensus protocol, devises an underlying DAG-based mempool for efficient transaction broadcasting. This mempool design is orthogonal to our work in this paper and suggests the potential of combining Narwhal and Wahoo as a promising direction for future research. It is worth noting that Wahoo's block retrieval mechanism shares a similar idea as the block pull mechanism in Narwhal. Both leverage certificates or proofs to ensure totality, thereby simplifying garbage collection through offloading blocks from prior rounds to a passive distributed store. On the other hand, Wahoo's block retrieval mechanism can be viewed as an upgraded version of Narwhal's block pull mechanism, catering to blocks with diverse tags.

C. Other Related Works

Despite concerns over liveness [38], a substantial body of research has concentrated on designing BFT consensus within partially-synchronous networks, whose representatives include FaB5 [37], SBFT [24], HotStuff [46], and Fast-HotStuff [26]. Similar to the goal of Wahoo, a work named BG [42] aims to reduce the communication overhead of partially-synchronous protocols. Specifically, within a modular framework, BG notably reduces the communication overhead of FaB5 and SBFT by an order of magnitude. Another work closely related to Wahoo is BBCA-CHAIN, which focuses on reducing the latency of DAG-based protocols [36]. Unlike Wahoo, which operates under the assumption of an asynchronous network, BBCA-CHAIN is designed for a partially-synchronous network.

A recent line of work considers mitigating block manipulation to enhance order fairness within consensus mechanisms. Pompe [48] utilizes timestamps for fair ordering, Aequitas [30] addresses the Condorcet paradox in transaction ordering via a notion of batch-order-fairness, and Themis [29] introduces a deferring technique to tackle the liveness issue in Aequitas. Despite their innovative approaches, these protocols rely on complex transaction graph processing, leading to efficiency issues. In contrast, ACCORD adopts an alternative strategy by designating multiple leaders to propose a block [4]. This method alleviates the risk of manipulation by a single leader and promises higher efficiency by eliminating the need for transaction graph analysis. These innovative protocols, including ACCORD, could be integrated into Wahoo, enhancing fairness in DAG-based protocols.

VIII. Conclusion

Current DAG-based BFT protocols excel in system efficiency by enabling parallel block broadcasting. However, they disregard the significant communication overhead. Addressing this, we introduce two new broadcast protocols,

PBC and EPBC, designed to exhibit linear communication overhead. Leveraging these protocols, we have devised Wahoo, a lightweight DAG-based BFT protocol that reduces communication overhead by an order of magnitude compared to existing solutions. Remarkably, under favorable situations where no replicas are faulty, Wahoo achieves low latency on par with the best results offered by existing protocols. Our experimental findings highlight Wahoo's exceptional performance in favorable conditions and its low communication overhead compared to its counterparts.

References

[1] I. Abraham, N. Ben-David, and S. Yandamuri, "Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2022, pp. 381–391.

[2] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Reaching consensus for asynchronous distributed key generation," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2021, pp. 363–373.

[3] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous Byzantine agreement," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2019, pp. 337–346.

[4] G. D. Bashar, J. Holmes, and G. G. Dagher, "ACCORD: A scalable multileader consensus protocol for healthcare blockchain," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 2990–3005, 2022.

[5] I. Bashir, *Mastering Blockchain*. Birmingham, U.K.: Packt, 2017.

[6] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *Proc. 2nd Annu. ACM Symp. Princ. Distrib. Comput.*, 1983, pp. 27–30.

[7] E. Blum, J. Katz, J. Loss, K. Nayak, and S. Ochsenreither, "Abraxas: Throughput-efficient hybrid asynchronous consensus," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2023, pp. 519–533.

[8] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.* Gold Coast, QLD, Australia: Springer, Nov. 2001, pp. 514–532.

[9] G. Bracha, "Asynchronous Byzantine agreement protocols," *Inf. Comput.*, vol. 75, no. 2, pp. 130–143, Nov. 1987.

[10] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. thesis, School Eng., Univ. Guelph, Guelph, ON, Canada, 2016.

[11] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Proc. Annu. Int. Cyptol. Conf.* Santa Barbara, CA, USA: Springer, 2001, pp. 524–541.

[12] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantipole: Practical asynchronous Byzantine agreement using cryptography (extended abstract)," in *Proc. 19th Annu. ACM Symp. Princ. Distrib. Comput.*, Jul. 2000, pp. 123–132.

[13] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd USENIX Symp. Operating Syst. Design Implement.*, 1999, pp. 173–186.

[14] X. Dai et al., "LightDAG: A low-latency DAG-based BFT consensus through lightweight broadcast," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2024, pp. 998–1008.

[15] X. Dai, B. Zhang, H. Jin, and L. Ren, "ParBFT: Faster asynchronous BFT consensus with a parallel optimistic path," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2023, pp. 504–518.

[16] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A DAG-based mempool and efficient BFT consensus," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 34–50.

[17] S. Das, Z. Xiang, and L. Ren, "Asynchronous data dissemination and its applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2021, pp. 2705–2721.

[18] S. Duan, M. K. Reiter, and H. Zhang, "BEAT: Asynchronous BFT made practical," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2028–2041.

[19] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[21] R. Friedman, A. Mostefaoui, and M. Raynal, "Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems," *IEEE Trans. Dependable Secure Comput.*, vol. 2, no. 1, pp. 46–56, Jan. 2005.

[22] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-NG: Fast asynchronous BFT consensus with throughput-oblivious latency," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 1187–1201.

[23] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Proc. Int. Conf. Financial Cryptography Data Secur.* Grenada: Springer, 2022, pp. 296–315.

[24] G. Golan Gueta et al., "SBFT: A scalable and decentralized trust infrastructure," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2019, pp. 568–580.

[25] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous BFT protocols," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 803–818.

[26] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, "Fast-HotStuff: A fast and resilient HotStuff protocol," 2020, *arXiv:2010.11454*.

[27] H. Jin and J. Xiao, "Towards trustworthy blockchain systems in the era of 'Internet of Value': Development, challenges, and future trends," *Sci. China Inf. Sci.*, vol. 65, pp. 1–11, May 2022.

[28] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2021, pp. 165–175.

[29] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in Byzantine consensus," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2023, pp. 475–489.

[30] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for Byzantine consensus," in *Proc. Annu. Int. Cryptol. Conf.*, 2020, pp. 451–480.

[31] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2020, pp. 1751–1767.

[32] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Princ.*, Oct. 2007, pp. 45–58.

[33] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[34] Y. Lu, Z. Lu, and Q. Tang, "Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined BFT," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 2159–2173.

[35] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-MVBA: Optimal multi-valued validated asynchronous Byzantine agreement, revisited," in *Proc. 39th Symp. Princ. Distrib. Comput.*, Jul. 2020, pp. 129–138.

[36] D. Malkhi, C. Stathakopoulou, and M. Yin, "BBCA-CHAIN: Low latency, high throughput BFT consensus on a DAG," 2023, *arXiv:2310.06335*.

[37] J.-P. Martin and L. Alvisi, "Fast Byzantine consensus," *IEEE Trans. Dependable Secure Comput.*, vol. 3, no. 3, pp. 202–215, Jul. 2006.

[38] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 31–42.

[39] A. Mostefaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous Byzantine consensus with $t<n/3$ and $o(n^2)$ messages," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2014, pp. 2–9.

[40] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 2705–2718.

[41] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distrib. Comput.*, vol. 2, no. 2, pp. 80–94, Jun. 1987.

[42] X. Sui, S. Duan, and H. Zhang, "BG: A modular treatment of BFT consensus toward a unified theory of BFT replication," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 44–58, 2024.

[43] Y. Xiao, N. Zhang, J. Li, W. Lou, and Y. T. Hou, "Distributed consensus protocols and algorithms," *Blockchain Distrib. Syst. Secur.*, vol. 1, pp. 25–55, Apr. 2019.

[44] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 2, pp. 1432–1465, 2nd Quart., 2020.

[45] X. Dai, Z. Zhang, J. Xiao, J. Yue, X. Xie, and H. Jin, "GradedDAG: An asynchronous DAG-based BFT consensus with lower latency," in *Proc. 42nd Int. Symp. Reliable Distrib. Syst. (SRDS)*, Sep. 2023, pp. 107–117.

[46] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2019, pp. 347–356.

[47] H. Zhang and S. Duan, "PACE: Fully parallelizable BFT from reproposable Byzantine agreement," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 3151–3164.

[48] Y. Zhang, S. T. V. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without Byzantine oligarchy," in *Proc. USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2020, pp. 633–649.

Xiaohai Dai (Member, IEEE) received the Ph.D. degree from the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), Wuhan, China, in 2021. He is currently a Post-Doctoral Researcher with the School of Computer Science and Technology, HUST. His current research interests include blockchain and distributed system. His awards include the Outstanding Creative Award in 2018 FISCO BCOS Blockchain Application Contest and Top Ten in FinTechathon 2019.

Zhaonan Zhang (Student Member, IEEE) received the bachelor's degree from Huazhong University of Science and Technology (HUST), Wuhan, China, in 2018. He is currently pursuing the master's degree with the School of Computer Science and Technology, HUST, under the supervision of Jiang Xiao. His research interests include blockchain and consensus.

Zhengxuan Guo (Student Member, IEEE) received the B.S. degree in computer science from Soochow University, China. He is currently pursuing the master's degree with Huazhong University of Science and Technology (HUST), under the supervision of Hai Jin. His research interests include blockchain and Byzantine fault tolerance.

Chaozheng Ding (Student Member, IEEE) received the B.S. degree in computer science from Yangtze University, China. He is currently pursuing the master's degree with Huazhong University of Science and Technology (HUST), under the supervision of Jiang Xiao. His research interests include blockchain and Byzantine fault tolerance.

Jiang Xiao (Member, IEEE) received the B.Sc. degree from HUST in 2009 and the Ph.D. degree from The Hong Kong University of Science and Technology (HKUST) in 2014. She is currently a Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), Wuhan, China. Her research interests include blockchain and distributed computing. Her awards include the CCF-Intel Young Faculty Research Program 2017, Hubei Downlight Program 2018, the ACM Wuhan Rising Star Award 2019, the Knowledge Innovation Program of Wuhan-Shuguang 2022, and the Best Paper Award from IEEE ICPADS/GLOBECOM/GPC/BLOCKCHAIN.

Xia Xie (Member, IEEE) received the Ph.D. degree in computer architecture from Huazhong University of Science and Technology in 2006. She is currently a Professor with Hainan University. Her research interests include data mining and knowledge graph.

Rui Hao (Member, IEEE) received the Ph.D. degree from Nanjing University (NJU), Nanjing, China, in 2023. She is currently a Post-Doctoral Researcher with the School of Computer Science and Artificial Intelligence, Wuhan University of Technology, China. Her research interests include software quality, software security, and blockchain.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), China, in 1994.

He was with The University of Hong Kong from 1998 to 2000 and a Visiting Scholar with the University of Southern California from 1999 to 2000. He is currently a Chair Professor of computer science and engineering with HUST. He has co-authored more than 20 books and published over 900 research articles. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security. He is a fellow of CCF and a Life Member of ACM. In 1996, he was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz in Germany and the Excellent Youth Award from the National Science Foundation of China in 2001.