Dory: Faster Asynchronous BFT with Reduced Communication for Permissioned Blockchains

Zongyang Zhang, *Member, IEEE*, You Zhou, Sisi Duan, *Member, IEEE*, Haibin Zhang, Bin Hu, Licheng Wang, *Member, IEEE*, Jianwei Liu, *Senior Member, IEEE*

***Abstract*—Asynchronous Byzantine fault-tolerance (BFT) protocols (e.g., HoneyBadger and Dumbo family protocols) have received increasing attention as the consensus mechanism of permissioned blockchains, given their particular robustness against timing and performance attacks. However, there is a substantial performance gap before they can be applied in real systems. In this paper, we identify and address two critical issues, and design Dory, an asynchronous BFT consensus protocol with improved efficiency and lower communication compared to the state-of-the-art protocol, sDumbo. At the core of our approach are two new building blocks reducing the communication cost and a novel framework utilizing transactions with quadratic message complexity.**

We have implemented Dory and sDumbo in a new Golang library. Via a deployment using up to 151 participants on Amazon EC2, we show that Dory consistently outperforms sDumbo during both failure and failure-free scenarios. For instance, Dory has up to 5x the throughput of sDumbo in the failure-free scenario.

***Index Terms*—Byzantine fault-tolerance, consensus, asynchronous, blockchain, communication complexity, fairness.**

I. Introduction

PERMISSIONED blockchains [1], [2] utilize Byzantine fault-tolerance (BFT) protocols [3]–[8] as their consensus mechanism to order transactions. Purely asynchronous BFT consensus protocols are gaining renewed attention due to their enhanced robustness without making any timing assumptions. This property makes them particularly suitable for the global deployment of blockchains, ensuring safety and liveness even in the presence of failures and unbounded network delays. However, there is a substantial performance gap before they can be applied in real systems, for both theoretical and practical reasons. In this paper, we identify and address two critical issues, and introduce Dory, a novel asynchronous BFT consensus protocol for permissined blockchain.

The communication bottleneck. Constructing an asymptotically optimal asynchronous BFT consensus protocol has been a long-standing challenge since the seminal work of Cachin et al. [9]. Given the number of participants n, input transaction size $|m|$, and the security parameter λ, the communication bound that one could theoretically hope for (so far) is $O(n^2|m| + \lambda n^2)$ [10]. Despite recent advancements [11], [12] in achieving optimal $O(n^2)$ message complexity, known instantiations (refer to Table I) still fall short of the expectation in terms of communication complexity. This theoretical gap leads to poor scalability in practice. Taking the state-of-the-art protocol, Speeding Dumbo (sDumbo), as an example, it has an $O(n^2|m| + \lambda n^3 \log n)$ communication complexity, where the cost of $O(\lambda n^3 \log n)$ that transfers redundant data (e.g., signatures and hashes) would expand much more rapidly than the effective payload cost of $O(n^2|m|)$ as the network size grows. Consequently, a bottleneck would emerge when deploying such a protocol in a large-scale network.

Wasted proposals. Another issue that impacts the performance of asynchronous BFT consensus is the potential "waste" of proposed transactions. Asynchronous BFT consensus protocols require all participants to propose in parallel and use an agreement component to elect and order the proposals. To tolerate the malicious behaviors of f faulty participants, the agreement component exclusively orders proposals from $n-f$ participants, although in the normal case, all participants may have proposed and successfully transmitted transactions. Namely, up to f valid proposals are un-delivered, and the relevant computation and bandwidth are "wasted."

Resolving the above issue is not straightforward in the asynchronous setting, as one cannot distinguish whether the missing of proposals is due to malicious behaviors or the network delay. Yang et al. propose a promising technique called *inter-node-linking* [13] to address this challenge. It allows one to temporarily set aside some proposals and re-use them later. However, it is only possible with expensive reliable broadcast protocols [12]–[15], resulting in high communication overhead and sub-optimal $O(n^3)$ message complexity.

Our contributions. Compared to prior works, we propose Dory, a faster asynchronous BFT consensus protocol with some distinguishing features: 1) achieving lower communication complexity and optimal message complexity; 2) resolving the wasted proposal issue; 3) achieving $5\times$ the throughput of the state-of-the-art. In addition, the design of Dory derives a new data dissemination primitive and a new implementation of agreement protocols, both of which might be of independent interest. Here we show our contributions in detail.

A new data dissemination primitive. We propose a new primitive called asynchronous *vector-data* dissemination (AVDD). It is a variant of Das et al.'s asynchronous data dissemination [18], with a focus on disseminating *vector-data* through the cooperation of participants. Besides, we also provide an

Zongyang Zhang, You Zhou, Bin Hu and Jianwei Liu are with School of Cyber Science and Technology, Beihang University, China. Emails: zongyangzhang, youzhou, hubin0205, liujianwei@buaa.edu.cn.

Sisi Duan is with Instititue for Advanced Study, Tsinghua University, China. Email: duansisi@tsinghua.edu.cn.

Haibin Zhang is with School of Computer Science and Technology, Beijing Institute of Technology, China. Email: haibin@bit.edu.cn.

Licheng Wang is with School of Cyber Science and Technology, Beijing Institute of Technology, China. Email: lcwang@bit.edu.cn.

Zongyang Zhang is the corresponding author.

TABLE I: Comparison for performance metrics. Communication complexity is measured in bits.

Protocol	Message Complexity	Communication Complexity[†]	Time Complexity
HoneyBadger [16]	$O(n^3)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$O(\log n)$
DispersedLedger [13]	$O(n^3)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$O(\log n)$
Dumbo [17]	$O(n^3)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$O(1)$
Speeding Dumbo (sDumbo) [11]	$O(n^2)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$O(1)$
Dory (this work)	$O(n^2)$	$O(n^2\|m\| + (\lambda + n)n^2 \log n)$	$O(1)$

[†] The security parameter λ denotes the length of signatures and hashes; in practice, λ is no less than 128. As $\lambda + n \ll \lambda n$, Dory has lower communication complexity than existing protocols.

efficient instantiation of the new primitive, which serves as a flexible recovery phase of our BFT consensus protocol and is crucial to lowering the communication complexity.

A new implementation of agreement protocol. Asynchronous BFT consensus protocols are typically randomized due to the FLP theorem [19]—there is no deterministic solution in the asynchronous setting. In the long-term effort to bypass this barrier, the recent works [11], [12], [20] using multi-valued validated byzantine agreement (MVBA) stand out because of better performance. In this paper, we also focus on MVBA-based protocols and implement an efficient MVBA protocol called dMVBA that achieves $O(n|v| + \lambda n^2 \log n)$ communication complexity for the first time.

A new BFT consensus framework utilizing proposals. We design a novel asynchronous BFT consensus framework that efficiently utilizes proposals and tackles the issue of wasted proposals. The framework develops the idea of *inter-node-linking* technique and seamlessly incorporates it with our AVDD and dMVBA. Different from the original work of Yang et al. [13], our framework relaxes the need for expensive broadcast primitives and is able to achieve the optimal $O(n^2)$ message complexity.

Reduced communication and improved performance. Finally, we propose Dory, an asynchronous BFT consensus protocol for permissioned blockchains, which makes progress both theoretically and practically.

- Dory is the first asynchronous BFT consensus protocol with $O(n^2|m| + (\lambda+n)n^2 \log n)$ communication complexity and optimal $O(n^2)$ message complexity, edging closer to the known bound [10]. Before us, the commonly known best communication complexity is $O(n^2|m| + \lambda n^3 \log n)$.
- Extensive experimental results show that Dory consistently outperforms sDumbo, the state-of-the-art protocol. We implement Dory and sDumbo in Golang[1], and evaluate them on up to 151 Amazon EC2 instances. Compared to sDumbo, Dory has significantly lower communication cost, preserves low latency (less than 8s) even for a large network, and achieves high throughput (up to 5× that of sDumbo at failure-free cases).

II. Preliminaries

A. Notations

Let $[n]$ denote the set of integers $\{1, 2, ..., n\}$. In our protocols, f out of n participants ($\{P_i\}_{i\in[n]}$) may be faulty. We may associate each protocol instance with a unique session identifier ID, tagging each message in the protocol with ID. More notations are listed in Table II.

[1]Our open-source code: https://github.com/xygdys/Dory-BFT-Consensus.

TABLE II: Notations

Notation	Description
n	Number of participants
f	Number of faulty participants
λ	Security parameter
e	Epoch number
P_i	The i-th participant
A	The global vector in AVDD
A_i	P_i's local version of A in AVDD
a_k	The k-th element of A/A_i in AVDD
$\|a\|$	Bit length of each element in A/A_i
R_i	P_i's requesting set in AVDD
d_{ki}	The i-th fragment of a_k in AVDD
T_k, T_k^*	Sets storing fragments of a_k
v_i	P_i's input of MVBA
$\|v\|$	Bit length of MVBA's each input
σ_i	Signature combined by P_i
ρ_{ij}	Share of σ_i signed by P_j
S	Set storing signature shares
rt_i	Root of P_i's Merkle tree
br_{ij}	The j-th branch of P_i's Merkle tree
$prop_i^e$	P_i's proposal of epoch e
m_i^e	P_i's input transactions of epoch e
$\|m\|$	Bit length of transactions in each proposal
h_i	Hash value of P_i's proposal
V	Common view vector
V_i	P_i's local view vector
ID	Unique session identifier of protocol instances

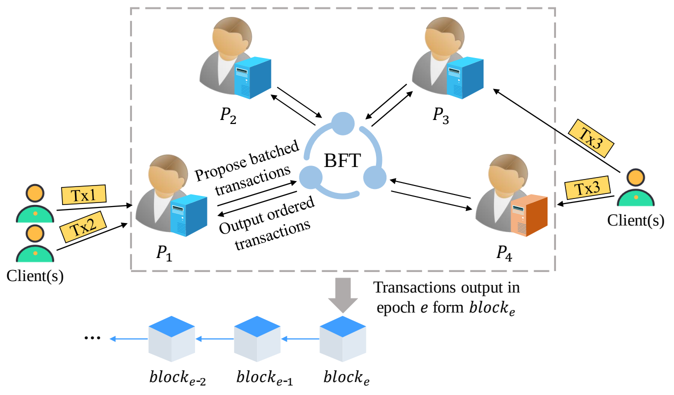
B. System Model

The Dory protocol proceeds in consecutive epochs, where in each epoch, participants agree on the order of a sequence of transactions submitted by clients and output them to form a block. Fig 1 illustrated an example of $n = 4$.

Client. Each client submits transactions to at least one of its trusted participants.

Participant. Each participant maintains an input buffer to store transactions submitted by clients. In each epoch, participants batch an identical size of transactions from its buffer to propose. Participants communicate via the BFT consensus and output ordered transactions at the end of each epoch. As in prior works [13], [21]–[23], we assume that correct participants have de-duplicated input buffers containing mostly different transactions.

Network. We consider completely asynchronous systems making no timing assumptions on message processing or transmission delays, and only assume peer-to-peer channels between participants.



P_2
P_3
BFT
Propose batched transactions
Output ordered transactions
P_1
P_4
Tx1
Tx2
Tx3
Tx3
Client(s)
Client(s)
Transactions output in epoch e form $block_e$
...
$block_{e-2}$
$block_{e-1}$
$block_e$

Fig. 1: System model of Dory.

Adversary. Our implementations tolerate static corruption, where the adversary needs to choose the set of faulty participants before the execution of the protocol. We focus on the Byzantine failure model where the faulty participants behave arbitrarily and $f \leq \lfloor \frac{n-1}{3} \rfloor$. A *quorum* is a set of $\lceil \frac{n+f+1}{2} \rceil$ participants. For simplicity, we may assume $n = 3f + 1$ and a quorum size of $2f + 1$.

Trusted setup. We assume trusted setup for the threshold cryptosystems.

C. Problem Definition

This paper studies asynchronous BFT consensus protocols, where participants *atomically outputs* transactions, each *being submitted* by some client. The correctness of BFT consensus protocols is specified as follows.

- *Agreement.* If any correct participant outputs a transaction m, then every correct participant outputs m.
- *Total order.* If a correct participant outputs a transaction m before outputting m', then no correct participant outputs a transaction m' without first outputting m.
- *Liveness.* If a correct client submits a transaction m to a correct participant, then every correct participant eventually outputs m with probability 1.
- *Fairness* (*Blockchain quality* [24]). Let S be the set of transactions that have been output by correct participants from the beginning of the protocol. If $S \neq \emptyset$, then at least $1/2$ of transactions in S were input by correct participants.

Note that the fairness property prevents an adversary from committing a majority of malicious transactions. It is also referred to other names, such as "blockchain quality" by Duan et al. [24].

D. Primitives

Multi-valued validated Byzantine agreement (MVBA). MVBA allows each participant that has an input to agree on a value v, which satisfies a global and polynomial-time computable $\mathcal{Q}$ known by all participants [9]. More formally, an MVBA protocol satisfies the following properties:

- *Agreement.* If any correct participant outputs v, then every correct participant outputs v.
- *External Validity.* If a correct participant outputs a value v, then v is valid, i.e., $\mathcal{Q}(v) = 1$.
- *Termination.* If $n - f$ correct participants have an input, then every correct participant gets an output with probability 1.

Threshold signature. Threshold signature allows any t participants to produce a valid signature, while any participants less than t cannot [25], [26]. It consists of five algorithms:

- *Key generation:* $\{pk, sk\} \leftarrow \mathsf{KeyGen}(\lambda, n, t)$. Given a security parameter λ, the total number of participants n and a threshold t, the algorithm outputs a public key pk, and a vector of secret keys $sk = (sk_1, sk_2, \dots, sk_n)$. For simplicity, pk is dropped for the following algorithms.
- *Signing:* $\rho_i \leftarrow \mathsf{Sign}_t(sk_i, m)$. Given a secret key sk_i, a message m, the algorithm outputs a signature share ρ_i.
- *Share verification:* $0/1 \leftarrow \mathsf{VerifyShare}_t(m, (i, \rho_i))$. Given a message m, an index i and a signature share ρ_i, the algorithm outputs 1 iff ρ_i is a valid signature share computed by participant P_i for m.
- *Combining:* $\sigma/\bot \leftarrow \mathsf{Combine}_t(m, \{(i, \rho_i)\}_{i\in S})$. Given a set of pairs $\{(i, \rho_i)\}_{i\in S}$, where $S \subset [n]$ and $|S| = t$, the algorithm outputs a signature σ iff all shares in S are valid.
- *Signature verification:* $0/1 \leftarrow \mathsf{Verify}_t(m, \sigma)$. Given a message m and a signature σ, the algorithm outputs 1 if σ is a valid signature for m; otherwise, it outputs 0.

A (n, t) threshold signature scheme should satisfy the conventional robustness and unforgeability properties.

Merkle tree. Merkle tree is a hash tree allowing position-binding commitment and verification on a vector. It consists of two algorithms:

- *Initialization:* $\{rt, br_1, br_2, \dots, br_n\} \leftarrow \mathsf{Merkle}(M)$. Given a n-dimensional vector $M = (m_1, m_2, \dots, m_n)$, the algorithm outputs a root rt, and a branch proof br_i for each $m_i, i \in [n]$.
- *Branch verification:* $0/1 \leftarrow \mathsf{VerifyBranch}(rt, i, m_i, br_i)$. Given a root rt, a position number i, an element m_i, and a branch proof br_i, the algorithm outputs 1 if m_i is the i-th element of the vector corresponding to rt; otherwise, it outputs 0.

Given a n-dimensional vector, the bit length of the root and the branch proof are $O(\lambda)$ and $O(\lambda \log n)$, respectively.

Provable broadcast (PB). PB is a broadcast protocol among n participants, where a designed participant (also called sender) with ID multicasts some m [9], [11], [27], [28]. Additionally, the sender will also output a tuple (h, σ) as proof, where h is the hash of m and σ is a threshold signature for h and ID. Formally, assuming a collision-resistant hash function $\mathcal{H}$, a PB protocol with an identifier ID satisfies the following properties:

- *Provability.* If the sender outputs any two tuples (h, σ) and (h', σ') s.t. $\mathsf{Verify}_{n-f}(\langle \mathsf{ID}, h \rangle, \sigma) = \mathsf{Verify}_{n-f}(\langle \mathsf{ID}, h' \rangle, \sigma') = 1$, then $h = h'$ and at least $f + 1$ correct participant output m s.t. $\mathcal{H}(m) = h$.
- *Termination.* If the sender is correct and inputs a value m, then all correct participants will output m. In addition, the sender will output (h, σ) satisfying $\mathsf{Verify}_{n-f}(\langle \mathsf{ID}, h \rangle, \sigma) = 1$ and $\mathcal{H}(m) = h$.

The PB protocol can be easily instantiated using a $(n, n - f)$ threshold signature, and achieving $O(n)$ messages and

$O(n|m| + \lambda n)$ communication [9], [11], [28].

Error correcting code. Error correcting code enables correcting errors or recovering missing fragments of the encoded data. It consists of the following algorithms:

- *Encode:* $\{d_1, d_2, \ldots, d_n\} \leftarrow \mathsf{Encode}(m, n, t)$. Given a data block m, which is split into t coefficients of a polynomial $p(\cdot)$ in a Galois Field $\mathbb{F}$, the algorithm encodes m to n fragments $\{d_1, d_2, \ldots, d_n\}$, where $d_i \in \mathbb{F}$ for $i \in [n]$.
- *Decode:* $m' \leftarrow \mathsf{Decode}(T, t, r)$. Given a set of fragments of T, some of which may be incorrect, the algorithm outputs a $t-1$ degree polynomial, i.e., a data block m', by correcting up to r errors in T.

It is well-known that the decode algorithm can successfully output the original data block provided $|T| \geq t+2r$ [29] (e.g., the Berlekamp-Welch algorithm [30], Gao's algorithm [31]).

III. Technical Overview

In this section, we outline the background, and then show our approach to building Dory, addressing the two issues stated in the introduction.

A. Background

Asynchronous BFT Consensus. The consensus problem was first introduced by Lamport et.al [32]. It has since been studied in various models with different assumptions. Recently, there has been renewed interest in BFT consensus protocols, driven by the advancements in blockchain. Pure asynchronous BFT consensus protocols are particularly favored due to their robustness against timing attacks.

The celebrated FLP theorem [19] shows that there is no deterministic consensus protocol in the asynchronous setting. Thus, asynchronous BFT consensus protocols typically use randomized agreement components to circumvent it. The current agreement protocols used in asynchronous BFT are predominantly classified into two primitives: asynchronous binary agreement (ABA) and multi-valued validated byzantine agreement (MVBA). The ABA-based protocols [13], [16], [33]–[36] have high complexities and perform well only when n is small. In contrast, the MVBA-based protocols [9], [11], [17] terminate in constant expected time and are more practical and scalable. This paper focuses on MVBA-based protocols.

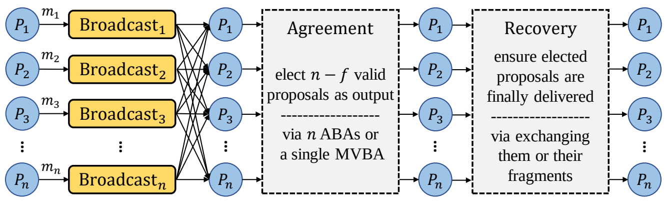

P_1
P_2
P_3
P_n
m_1
m_2
m_3
m_n
Broadcast$_1$
Broadcast$_2$
Broadcast$_3$
Broadcast$_n$
Agreement
elect $n-f$ valid proposals as output
via n ABAs or a single MVBA
Recovery
ensure elected proposals are finally delivered
via exchanging them or their fragments

Fig. 2: A general construction of asynchronous BFT.

Fig. 2 shows a general construction of asynchronous BFT consensus (e.g., sDumbo [11]), including three phases in each epoch. 1) In the broadcast phase, all n participants batch their transactions into proposals and send them via broadcast component in parallel. 2) The election phase determines $n-f$ valid proposals as the output of this epoch. If using MVBA, a common approach to electing proposals is letting each participant collect $n-f$ proofs of received proposals, and input the set of proofs to MVBA; then MVBA outputs one participant's set, indicating $n-f$ proposals are elected. 3) The recovery phase is used to ensure the elected proposals will delivered by all correct participants. It is useful as up to f elected proposals may be from faulty participants. Many recent works follow this (or a similar) construction [11]–[13].

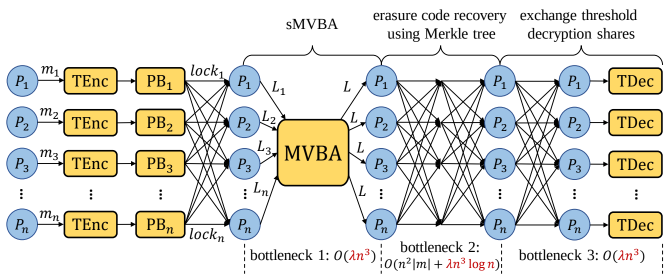

sMVBA
erasure code recovery using Merkle tree
exchange threshold decryption shares
P_1
P_2
P_3
P_n
m_1
m_2
m_3
m_n
TEnc
PB$_1$
PB$_2$
PB$_3$
PB$_n$
$lock_1$
$lock_n$
L_1
L_2
L_3
L_n
MVBA
L
TDec
bottleneck 1: $O(\lambda n^3)$
bottleneck 2: $O(n^2|m| + \lambda n^3 \log n)$
bottleneck 3: $O(\lambda n^3)$

Fig. 3: The sDumbo framework and its communication bottleneck ingredients.

sDumbo. In the construction of asynchronous BFT consensus protocols, various ingredients contribute to inefficiencies, particularly in terms of communication complexity. We take the state-of-the-art work, sDumbo [11], as an example. Fig. 3 illustrates the construction of sDumbo, which includes a PB (provable broadcast)-based broadcast phase, an MVBA (sMVBA)-based election phase, a recovery phase recovering missing proposals, and a threshold decryption operation. The election phase, the recovery phase, and the threshold decryption operation have $O(\lambda n^3)$, $O(n^2|m| + \lambda n^3 \log n)$, and $O(\lambda n^3)$ communication, respectively. Note that the recovery phase of sDumbo requires participants to exchange $O(n^3)$ Merkle tree proofs each of which is $O(\lambda \log n)$ bits, and thus incurs a high redundant communication. However, it cannot be omitted as it plays a crucial role in ensuring the agreement property. Indeed, many recent protocols [11]–[13], [22] are faced with the same issue that the recovery phase is the most expensive ingredient in their design.

sDumbo follows the general construction illustrated in Fig. 2, but with an additional threshold encryption technique to preserve liveness. In particular, the encrypted proposals prevent the adversary from censoring specific transactions and making the system violate the liveness property. However, there are only $n-f$ proposals can be output in each epoch, and thus the remaining are still wasted. That is, achieving liveness does not imply resolving the wasted proposal issue.

Inter-node-linking. Yang et al. [13] propose a promising technique called *inter-node-linking* to solve the wasted proposals issue. The basic idea is to let each participant not only input the proposal of the current epoch but also embed an observation about the proposals of previous epochs; then, participants make an agreement on proposals of both the current epoch and previous epochs. That is, if some valid proposal was discarded in the epoch that it was initialized, it could still be utilized and output in a later epoch. Fig. 4 illustrates an example of this, where each participant maintains a view vector V_i representing the observation on previous proposals. In particular, if P_i has

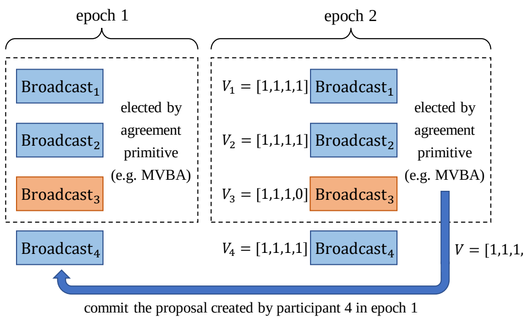


Fig. 4: An example of inter-node-linking. Participant P_3 are faulty while the others are correct. The proposal of P_4 in epoch 1 is not elected but is still committed and output in epoch 2 as all correct participants have recorded it is valid in their view vectors.

received P_j's proposals for all epochs before e, P_i will set $V_i[j]$ as e.

However, the *inter-node-linking* technique inherently requires expensive broadcast primitives, e.g., reliable broadcast, and n parallel instances of known practical reliable broadcast [15], [18], [37] incur at least $O(\lambda n^3)$ redundant communication and $O(n^3)$ messages complexity. Therefore, the framework of Yang et al. results in inefficiency in terms of both message complexity and communication complexity.

B. Our Approach

In the design of Dory, we have two goals in mind: 1) reducing the communication; and 2) resolving the wasted proposal issue. First, we carefully design and implement two building blocks, including AVDD and dMVBA, which are crucial to lowering communication complexity. Then, we propose a novel asynchronous BFT consensus framework that is efficiently incorporated with our new building blocks to utilize proposals. Here we show a nutshell of our approach.

AVDD. As discussed in Sec.III-A, many ingredients of current protocols lead to high communication complexity. In particular, the recovery phase is uniformly the most expensive. We propose a new primitive called asynchronous *vector-data* dissemination (AVDD). It is able to disseminate a vector to all participants through the cooperation of them. Formally, suppose that we have a vector with that any element has been held and verified by at least $f+1$ correct participants, then AVDD allows every correct participant to obtain the same vector. We also provide an efficient instantiation which incurs $O(\ell n|a| + \ell n^2 \log n)$ communication complexity and $O(n^2)$ message complexity for an ℓ-dimensional vector $A = (a_1, \ldots, a_\ell)$.

AVDD naturally serves as a recovery phase of asynchronous BFT consensus, as the elected proposals collectively form a vector, with each proposal held and verified by at least $f+1$ correct participants upon its election. In particular, for the recovery of $O(n)$ $|m|$-size proposals, $\ell = O(n)$ and $|a| = |m|$, then our instantiation incurs $O(n^2|m| + n^3 \log n)$ communication complexity, achieving an $O(\lambda)$ reduction compared to existing works. Further, we utilize a technique enabling our

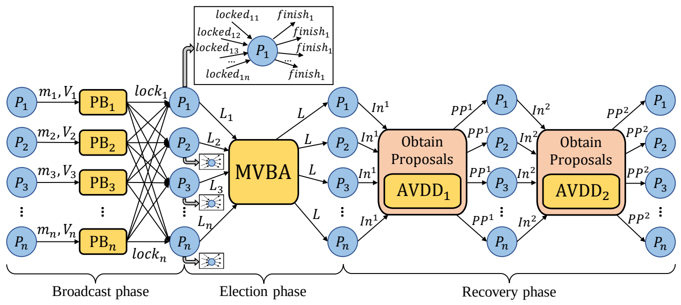


Fig. 5: The Dory framework.

instantiation to run in an *on-request* manner which makes it more efficient in the optimistic case, with a nearly *zero* communication overhead.

dMVBA. The MVBA protocol used in existing works is another bottleneck ingredient, leading to a communication complexity of at least $O(\lambda n^3)$. This paper addresses this issue by implementing a more efficient MVBA protocol, building upon recent advancements in various performance aspects of MVBA [10], [11]. Specifically, we use the APDB technique from Lu et al. [10] on sMVBA [11] to reduce the communication complexity, and thus instantiate an MVBA protocol named dMVBA with $O(n|v| + \lambda n^2 \log n)$ communication complexity. If applied to asynchronous BFT consensus protocols, dMVBA only incurs $O(\lambda n^2 \log n)$ communication complexity assuming $|v| = O(n\lambda)$.

The Dory framework. If we incorporate the above building blocks in the framework of sDumbo, only the communication of the election and recovery phases are reduced, whereas the threshold decryption still consumes $O(\lambda n^3)$ bits of communication. Besides, the wasted proposal issue has not been addressed. Also, the framework proposed by Yang et al. [13] results in high message complexity and communication complexity, which cannot be solved by simply replacing underlying building blocks. Therefore, we need new design.

We propose a more efficient framework in this paper, which resolves the wasted proposal issue with optimal message complexity and lower communication. It has developed the basic idea of *inter-node linking* [13] with much different and advanced technical designs. Here we show an overview of the Dory framework.

As shown in Fig. 5, it consists of three phases: broadcast, election and recovery. Different from prior works, the execution of these phases in the Dory framework has two parallel forward paths: one *normal path* to agree on most current proposals and one *supplemental path* to utilize previously undelivered proposals.

In the *normal path*, each participant invokes the PB instance with its proposal as input to produce a *lock* proof and broadcast it. Upon collecting $n-f$ *lock* proofs from other participants, each participant enters the election phase and input the *lock* proofs into MVBA. Then, with the help of MVBA and the first AVDD instance, participants are able to agree on and deliver $n-f$ proposals of the current epoch.

The *supplemental path* is executed simultaneously with the election phase of the normal path. After receiving the *lock*

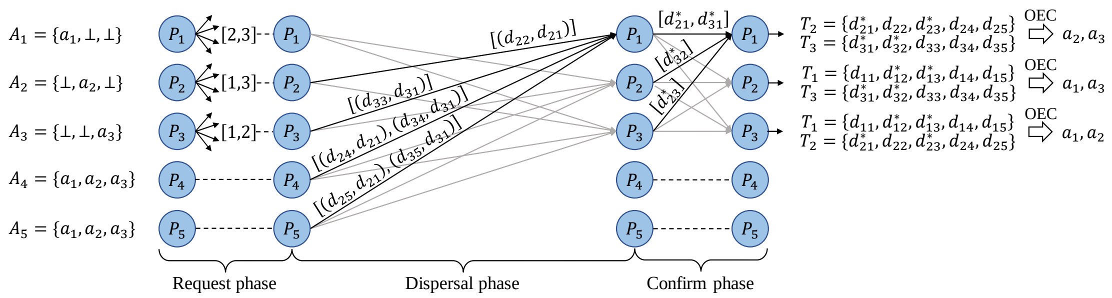

$A_1 = \{a_1, \perp, \perp\}$
$A_2 = \{\perp, a_2, \perp\}$
$A_3 = \{\perp, \perp, a_3\}$
$A_4 = \{a_1, a_2, a_3\}$
$A_5 = \{a_1, a_2, a_3\}$
P_1 P_2 P_3 P_4 P_5
[2,3]
[1,3]
[1,2]
$[(d_{22}, d_{21})]$
$[(d_{33}, d_{31})]$
$[(d_{24}, d_{21}), (d_{34}, d_{31})]$
$[(d_{25}, d_{21}), (d_{35}, d_{31})]$
$[d^*_{21}, d^*_{31}]$
$[d^*_{32}]$
$[d^*_{23}]$
$T_2 = \{d^*_{21}, d_{22}, d^*_{23}, d_{24}, d_{25}\}$
$T_3 = \{d^*_{31}, d^*_{32}, d_{33}, d_{34}, d_{35}\}$
OEC
a_2, a_3
$T_1 = \{d_{11}, d^*_{12}, d^*_{13}, d_{14}, d_{15}\}$
$T_3 = \{d^*_{31}, d^*_{32}, d_{33}, d_{34}, d_{35}\}$
OEC
a_1, a_3
$T_1 = \{d_{11}, d^*_{12}, d^*_{13}, d_{14}, d_{15}\}$
$T_2 = \{d^*_{21}, d_{22}, d^*_{23}, d_{24}, d_{25}\}$
OEC
a_1, a_2
Request phase
Dispersal phase
Confirm phase

Fig. 6: An example of the AVDD protocol where $n = 7$ and $f = 2$. Faulty participants are omitted. In the request phase, P_1 requests its missing elements. After the dispersal phase, P_1 will receive $f+1 = 3$ consistent fragments for a_2, a_3 respectively, and hence set d^*_{21} and d^*_{31}. Similarly, P_2 will set d^*_{12}, d^*_{32}, P_3 will set d^*_{13}, d^*_{23}. In the confirm phase, P_1 will receive d^*_{21}, d^*_{23} for a_2 and d^*_{31}, d^*_{32} for a_3. Thus, P_1 is able to collect $2f+1 = 5$ correct fragments for a_2, a_3 and successfully reconstruct them. Similarly, other participants also reconstruct their missing elements.

proof of a PB instance, each participant creates a partial signature and sends it to the sender. The sender combines the signature shares from the participants into a *finish* proof and broadcasts the proof. Then, each correct participant that receives the *finish* proof is able to report the corresponding proposal in a view vector that will be included in later proposals. In a later epoch, if a proposal including the view vector is delivered through the normal path, then an additional AVDD instance of the recovery phase will be triggered to deliver the proposals indexed in the view vector.

The key to our design is the *supplemental path* which stems from a critical observation: the AVDD primitive can efficiently reconstruct an incomplete vector of proposals for all correct participants, as long as the proof of each PB is formed and verified by at least $f+1$ participants. Accordingly, upon a *finish* proof of some discarded proposal is formed, then all participants could obtain this proposal through AVDD, and thus it can be securely utilized.

By combining our new framework with new building blocks, e.g. instantiating MVBA with dMVBA, we finally obtain the Dory protocol, which resolves the wasted proposal issue and achieves $O(n^2)$ message complexity and $O(n^2|m| + (\lambda+n)n^2 \log n)$, lower than $O(n^2|m| + \lambda n^3 \log n)$ of the existing works.

IV. Designed Building Blocks

In this section, we introduce the asynchronous BFT building blocks by design, including asynchronous vector-data dissemination (AVDD) and a new implementation of multi-valued validated Byzantine agreement (dMVBA).

A. Asynchronous Vector-data Dissemination

In this paper, we first aim to design a more efficient recovery phase for asynchronous BFT consensus protocols. Therefore, we define *asynchronous vector-data dissemination* as a first-class primitive in the following.

Asynchronous vector-data dissemination (AVDD). In a distributed system with n participants, suppose that we have a global ℓ-dimensional vector $A = (a_1, \ldots, a_\ell)$ and for any $k \in [\ell]$, at least $f+1$ correct participants hold the same a_k and verify its correctness; then AVDD is to allow every correct participant to obtain the same A. Formally, it satisfies the following properties:

- *Validity.* For any $k \in [\ell]$, if at least $f+1$ correct participants hold the same $a_k \neq \perp$ and other correct participants set $a_k = \perp$, then every correct participant outputs a complete vector A with no $\perp$ element.
- *Consistency.* For any two correct participants P_i and P_j, if they output A_i and A_j separately, then $A_i = A_j$.

For this new primitive, we provide a simple and efficient instantiation, but with some skillful designs. The basic idea is to run the protocol in an *on-request* manner, i.e., each participant initially examines its local version of the global vector, requests the missing elements from others, and then other participants reply with the requested elements. The key to this idea is determining how and when participants reply. First, we use error correcting code to reduce the communication cost and thus participants only send encoded fragments instead of whole elements. However, such a technique leads to a liveness risk: if each participant only replies with fragments of elements that it holds, then the requesting participant may not receive enough fragments to decode the missing elements; or if each participant does not reply until holds all the requested elements, there may be a deadlock issue that every participant waits fragments from others at the same time and the protocol is stuck. To eliminate this liveness risk, we design a *divide and conquer* technique. It allows each participant to divide each request into two parts and conquer them separately: for the requested elements that the participant holds, it replies with the fragments immediately; while for those that the participant does not hold, it defers the reply until receiving the corresponding fragments from other participants.

Fig. 6 shows an example of the AVDD protocol, and now we briefly introduce how a request is processed. It consists of three phases: request, dispersal, and an optional confirm phase. In the request phase, each participant creates a request to claim the missing elements. In the dispersal phase, upon receiving a request, every participant sends the fragments of elements that it holds (if any). For those requested elements that the participant does not hold, it waits until receiving the corresponding fragments from other participants, then in the confirm phase passes them to the participant that initiated the

Algorithm 1 AVDD protocol with identifier ID for P_i

```
1: Initialization: $R_i \leftarrow \{\}$; $ReadyFlag \leftarrow false$; for $k \in [\ell]$ do, $d^*_{ki} \leftarrow \perp, T_k \leftarrow \{\}, T^*_k \leftarrow \{\}$
2: upon receiving input $A_i = (a_1, a_2, \dots, a_\ell)$ do
3:     for $1 \le k \le \ell$ do
4:         if $a_k = \perp$ then
5:             $R_i \leftarrow R_i \cup \{k\}$
6:     if $R_i$ is not empty then
7:         broadcast $(\text{REQUEST}, \mathsf{ID}, R_i)$                ▷ Request phase
8:     $ReadyFlag \leftarrow true$

9: wait until $ReadyFlag = true$
10:    upon receiving $(\text{REQUEST}, \mathsf{ID}, R_j)$ from $P_j$ do
11:        $D \leftarrow \{\}$, $C \leftarrow \{\}$
12:        for any $k \in R_j$ and $a_k \ne \perp$ do                 ▷ Disperse phase
13:            $(d_{k1}, d_{k2}, \dots, d_{kn}) \leftarrow \mathsf{Encode}(a_k, n, f+1)$
14:            $D \leftarrow D \cup (k, d_{ki}, d_{kj})$
15:        send $(\text{DISPERSE}, \mathsf{ID}, D)$ to $P_j$
16:        for any $k \in R_j$ and $a_k = \perp$ do
17:            wait until $d^*_{ki} \ne \perp$                        ▷ Updated in ln 26
18:            $C \leftarrow C \cup (k, d^*_{ki})$
19:        if $C$ is not empty then
20:            send $(\text{CONFIRM}, \mathsf{ID}, C)$ to $P_j$          ▷ Confirm phase
21:    upon receiving $(\text{DISPERSE}, \mathsf{ID}, D)$ from $P_j$ do
22:        for any $(k, d_{kj}, d_{ki}) \in D$ do
23:            $T_k \leftarrow T_k \cup \{(j, d_{kj})\}$
24:            $T^*_k \leftarrow T^*_k \cup \{(j, d_{ki})\}$
25:            if there are $f+1$ consistent $(\cdot, d_{ki})$ in $T^*_k$ then
26:                $d^*_{ki} \leftarrow d_{ki}$
27:    upon receiving $(\text{CONFIRM}, \mathsf{ID}, C)$ from $P_j$ do
28:        for any $(j, d_{kj}) \in C$ do
29:            $T_k \leftarrow T_k \cup \{(j, d_{kj})\}$

30:    for any $k \in R_i$ do
31:        upon $|T_k| \ge 2f+1$ do                              ▷ Trigger OEC for $m_k$
32:            for $0 \le r \le f$ do
33:                wait until $|T_k| \ge 2f+r+1$
34:                $p_k(\cdot) \leftarrow \mathsf{Decode}(T_k, f+1, r)$
35:                if $2f+1$ $(j, y) \in T_k$ satisfy $p_k(j) = y$ then
36:                    $a_k \leftarrow$ coefficients of $p_k(\cdot)$

37:    wait until no element of $A_i$ is $\perp$
38:        output $A_i$ as $A$
```

request. Finally, each participant that initiated a request will collect enough fragments and run the online error correcting (OEC) [33] algorithm to reconstruct the missing elements.

We now present our instantiation (AVDD protocol) in detail. The pseudocode for P_i is shown in Algorithm 1.

Initialization. Every participant P_i begins with an input A_i, which is a vector of ℓ elements, i.e., $a_1, \dots, a_\ell$. Depending on the protocol that triggers the AVDD protocol, some elements might be $\perp$. Participant P_i initializes several global parameters: a set R_i tracking the elements that need to be reconstructed; for each $k \in [\ell]$, a value d^*_{ki} and two sets T_k and T^*_k storing the data fragments.

Request. At the beginning of the protocol, P_i first checks A_i and adds k to a set R_i if $a_k = \perp$. If R_i is not empty, P_i broadcasts a $(\text{REQUEST}, \mathsf{ID}, R_i)$ message to all participants (ln 2-8) and then waits for all the elements in A_i to become non-empty (ln 37).

Dispersal. If P_i receives an incoming REQUEST message from participant P_j, it checks A_i and initializes two sets, D and C. D is used to store a set of fragments for any $k \in R_j$ and a_k is not $\perp$. C is used to store a set of fragments for $k \in R_j$ and a_k is $\perp$. In our protocol, P_i can directly update D and send a set of fragments to P_j, while the update of C might be deferred but will eventually be completed. Specifically, we distinguish two cases for each participant P_i:

- Case 1: for any $k \in R_j$ that $a_k \ne \perp$, P_i encodes a_k to obtain the i-th and j-th data fragments and adds a tuple (k, d_{ki}, d_{kj}) to D. After that, P_i sends a $(\text{DISPERSE}, \mathsf{ID}, D)$ message to P_j (ln 12-15).
- Case 2: for any $k \in R_j$ such that $a_k = \perp$, P_i waits until d^*_{ki} is updated. Namely, P_i need to obtain the corresponding fragments from other participants before replying to the request, which is handled in the confirm phase.

Confirm. For each elements P_i requests, upon receiving a $(\text{DISPERSE}, \mathsf{ID}, D)$ message from P_j, P_i adds the data fragments to the T_i and T^*_i sets. If there are $f+1$ matching fragments d_{ki}, P_i sets d^*_{ki} as d_{ki}. As we show in our proof, for every element P_i requests in R_i, it will receive at least $f+1$ matching d_{ki} from other participants (ln 21-26). Recall that some requests could be not totally handled in the dispersed phase (case 2). For such a request with R_j, if all d^*_{ki} are updated for any $a_k = \perp$ and $k \in R_j$, P_i adds them to C and sends a CONFIRM message to P_j (ln 16-20).

If case 2 is triggered, the participant that send a request may receive CONFIRM messages. Upon receiving a CONFIRM message, P_i adds the fragments to T_i (ln 27-29).

Upon collecting $2f+1$ fragments for any $k \in R_i$, P_i triggers the online error correcting (OEC) algorithm [33] to reconstruct a_k. Concretely, each execution of the OEC algorithm performs up to f trials of reconstruction. The number of required fragments increases with the number of trials. As the f^{th} trial satisfies $|T_k| \ge 3f+1$, P_i eventually reconstruct a_k, as mentioned in Section II-D (ln 30-36). Finally, P_i waits until it reconstructs all the elements such that there is no $\perp$ in A_i. Then P_i outputs A_i.

Correctness. We prove Algorithm 1 satisfies the two properties of our AVDD primitive in Sec. VI-A.

Complexity. Algorithm 1 has an $O(n^2)$ messages complexity as it only involves all-to-all communication. Its communication cost is *zero* in the optimistic case where all participants are correct and none of them miss any elements, and at most $4\ell n|a| + O(\ell n^2 \log n)$ in the worst case. Thus, the communication complexity is $O(\ell n|a| + \ell n^2 \log n)$. Please refer to Sec. VI-A for detailed analysis.

Comparison with ADD. In the design of AVDD, we are inspired by the asynchronous data dissemination (ADD) proposed by Das et al. [18]. Compared with their work, our primitive and instantiation have several distinguishing features: 1) focusing on the data dissemination of a vector instead of a single data block, which makes it more well-fitting for asynchronous BFT consensus protocols; 2) running in an *on-request* manner, allowing participants to exchange information of missing data only. As a result, our AVDD protocol achieves optimal $O(n^2)$ message complexity and lower communication overhead. Table III shows the performance metrics of AVDD and ADD. As the ADD protocol only disseminate a single data block at once execution, we use ℓ-ADD representing ℓ ADD instances for a fair comparison.

Note that AVDD has a significantly lower communication

TABLE III: Comparison between AVDD and ℓ-ADD

Protocol	**Message Complexity**	**Communication Cost**	
		Optimistic Case	Worst Case
AVDD	$O(n^2)$	0	$4\ell n\|a\|+O(\ell n^2 \log n)$
ℓ-ADD	$O(\ell n^2)$	$6\ell n\|a\| + O(\ell n^2 \log n)$	

cost on the payload ($0 \sim 4\ell n|a|$ v.s. $6\ell n|a|$), which has been demonstrated crucial for achieving high throughput of asynchronous BFT consensus protocols [11].

B. New Multi-valued Validated Byzantine Agreement

The MVBA protocol used in current BFT protocols is another bottleneck ingredient. For example, the sMVBA [11] protocol used in sDumbo has an $O(n^2|v|+\lambda n^2)$ communication complexity. If applying it in the election phase, there will be $|v| = O(\lambda n)$, as each participant collects $O(n)$ proofs of received proposals as the input (refer to Fig. 2 in Sec. III-A). Therefore, it incurs $O(\lambda n^3)$ communication cost.

In this paper, we address this issue by implementing a new MVBA protocol with lower communication complexity. We adopt the APDB technique proposed by Lu et al. [10] on sMVBA [11] to reduce the communication complexity, and thus construct a new MVBA protocol named dMVBA. In technical, dMVBA drives sMVBA as a block-box: at first, each participant encodes its input and disperses the fragments, then it collects replies from other participants and thus forms a proof; after that, each participant invokes sMVBA with the short proof instead of the input; finally, participants will exchange their fragments to reconstruct the original data according to the proof output by sMVBA.

We now describe dMVBA in detail. Algorithm 2 and Fig. 7 show the pseudocode and the workflow, respectively.

Details of dMVBA. At the beginning, participant P_i encodes its input into fragments, computes a Merkle tree over them, and send each P_j an ECHO message with the corresponding fragment and Merkle tree branch br_{ij}. Then, upon receiving a valid fragment and branch, each participant returns a signature share for the Merkle root rt_i via a READY message. After collecting $n-f$ signature shares, P_i combines them into a signature σ_i and triggers sMVBA with rt_i and σ_i. Upon sMVBA output (k, rt_k, σ_k), participants exchange fragments to reconstruct v'_k. Finally, P_i re-computes the Merkle tree on v'_k, then checks whether it has been correctly encoded (i.e., $rt'_k = rt_k$), and satisfies the predicate. If true, output v'_k; otherwise, P_k must be faulty and re-invoke sMVBA.

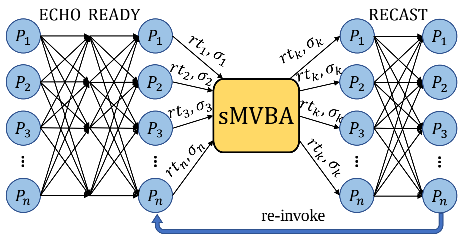


Fig. 7: The workflow of dMVBA protocol.

Correctness. The correctness of dMVBA follows [10]. We present a security intuition in this paper. For the agreement

Algorithm 2 dMVBA protocol with identifier ID and global known predicate $\mathcal{Q}$. Code shown for P_i

```
**Initialization:** $S \leftarrow \{\}$
let the $\mathcal{Q}'$ of the underlying sMVBA$[\langle \mathsf{ID}, r \rangle]$ be the following predicate:
$\mathcal{Q}'(k, rt_k, \sigma_k) \equiv (\mathsf{Verify}_{n-f}(\langle \mathsf{ID}, k, rt_k \rangle, \sigma_k) = 1)$

upon receiving input $v_i$ s.t. $\mathcal{Q}(v_i) = 1$ do
    $(d_{i1}, d_{i2}, \ldots, d_{in}) \leftarrow \mathsf{Encode}(v_i, n, f+1)$
    $(rt_i, br_{i1}, br_{i2}, \ldots, br_{in}) \leftarrow \mathsf{Merkle}(d_{i1}, d_{i2}, \ldots, d_{in})$
    for any $j \in [n]$ do
        send (ECHO, $\mathsf{ID}, d_{ij}, rt_i, br_{ij}$) to $P_j$

upon receiving (ECHO, $\mathsf{ID}, d_{ji}, rt_j, br_{ji}$) from $P_j$ do
    if $\mathsf{VerifyBranch}(rt_j, i, d_{ji}, br_{ji}) = 1$ do
        $\rho_{ji} \leftarrow \mathsf{Sign}_{n-f}(sk_i, \langle \mathsf{ID}, j, rt_j \rangle)$
        send (READY, $\mathsf{ID}, \rho_{ji}$) to $P_i$
        store $d_{ji}, br_{ji}$

upon receiving (READY, $\mathsf{ID}, \rho_{ij}$) from $P_j$ do
        if $\mathsf{VerifyShare}_{n-f}(\langle \mathsf{ID}, i, rt_i \rangle, (j, \rho_{ij})) = 1$ then
            $S \leftarrow S \cup \{j, \rho_{ij}\}$
            if $|S| = n - f$ then
                $\sigma_i \leftarrow \mathsf{Combine}_{n-f}(\langle \mathsf{ID}, i, rt_i \rangle, S)$

wait until $\sigma_i \neq \perp$
    for $r \in \{1, 2, 3, \ldots\}$ do
        invoke sMVBA$[\langle \mathsf{ID}, r \rangle]$ with input $(i, rt_i, \sigma_i)$
        wait until receiving $(k, rt_k, \sigma_k)$ from sMVBA$[\langle \mathsf{ID}, r \rangle]$ do
            $T \leftarrow \{\}$
            if $d_{ki} \neq \perp$ then
                broadcast (RECAST, $\mathsf{ID}, d_{ki}, br_{ki}$)
            upon receiving (RECAST, $\mathsf{ID}, d_{kj}, br_{kj}$) from $P_j$ do
                if $\mathsf{VerifyBranch}(rt_k, j, d_{kj}, br_{kj}) = 1$ do
                    $T \leftarrow T \cup \{(j, d_{kj})\}$
            upon $|T| = f + 1$ do
                $v'_k \leftarrow \mathsf{Decode}(T, f+1, r)$
                $(rt'_k, \cdot) \leftarrow \mathsf{Merkle}(\mathsf{Encode}(v`_k, n, f+1))$
                    if $rt'_k = rt_k$ and $\mathcal{Q}(v'_k) = 1$ then
                        output $v'_k$ and break the loop
```

property, the underlying sMVBA ensures participants receive the same (k, rt_k, σ_k) in each loop, and the Merkle tree ensures the consistency of fragments received by correct replicas: if P_k is correct, they will decode the same valid v'_k and output it; otherwise, they will find $rt'_k \neq rt_k$ and go back together. For the termination property, sMVBA ensures there is at least $1/2$ probability of electing a (k, rt_k, σ_k) that P_k is correct, and thus dMVBA terminates after the expected two executions of sMVBA. For the external validity property, it is straightforward that only the value satisfies $\mathcal{Q}$ can be output.

Complexity. Algorithm 2 only involves all-to-all communication and thus its message complexity is $O(n^2)$. The size of ECHO message, READY message, and RECAST message are $O(\frac{|v|}{f} + \lambda \log n)$, $O(\lambda)$ and $O(\frac{|v|}{f} + \lambda \log n)$, respectively. Further, the underlying sMVBA incurs $O(\lambda n^2)$ communication cost as its input size is $O(\lambda)$. Therefore, the communication complexity of dMVBA is $O(n|v| + \lambda n^2 \log n)$. The time complexity of dMVBA is $O(1)$ as it terminates in the expected constant rounds.

V. The Dory Framework and Protocol

This section presents the design of the Dory framework and protocol. We start by introducing some definitions.

Status of proposals. In Dory, the status of each proposal would transition according to the progress of participants. In

Algorithm 3 Utility functions of Dory. Code shown for P_i.

```
 1: procedure UpdateView($e$):
 2:     initialize a $|n|$-dimensional vector $V_i$
 3:     for any $j \in [n]$ do
 4:         $V_i[j] \leftarrow$ the latest $e'$ s.t. $e' < e$ and $finish^{e''}[j] = 1$ for all $1 \leq e'' \leq e'$
 5:     return $V_i$

 6: procedure ObtainProposals($\mathsf{ID}, In$):
 7:     initialize a $|In|$-dimensional vector $PP_i$, set $k \leftarrow 0$
 8:     for any $(e, j) \in In$ do
 9:         if $lock^e[j] = 1$ then
10:             $PP_i[k] = prop_j^e$
11:         else
12:             $PP_i[k] = \perp$
13:         $k \leftarrow k + 1$
14:     invoke AVDD[$\mathsf{ID}$] with input $PP_i$
15:     wait until AVDD[$\mathsf{ID}$] outputs $PP$
16:         for $(e, j) \in T$ do
17:             $lock^e[j] \leftarrow 1$, $finish^e[j] \leftarrow 1$, $commit^e[j] \leftarrow 1$
18:         return $PP$

19: procedure CheckViews($views, In$):                ▷ Refer to Equation 1
20:     initialize a $n$-dimensional vector $V$
21:     for any $j \in [n]$ do
22:         $V[j] \leftarrow$ the $(f+1)^{th}$ largest value among $\{V_k[j] | V_k \in views\}$
23:     for any $j \in [n]$ and $1 \leq e \leq V[j]$ do
24:         if $commit^e[j] = 0$ then
25:             $In \leftarrow In \cup \{(e, j)\}$
```

particular, the status of $prop_j^e$ (proposal created by P_j in epoch e) maintained by participant P_i can be one of the following:

- *locked.* If P_i has received a proposal $prop_j^e$ from P_j and receives a *lock* proof (h_j, σ_j) where σ_j is a valid signature for $\langle e, j, h_j \rangle$ and $h_j = \mathcal{H}(prop_j^e)$, then the status is *locked* and P_i sets $lock^e[j]$ as 1.
- *finished.* If P_i receives a proof σ_j' in a FINISH message for the proposal $prop_j^e$, then the status is *finished* and P_i sets $finish^e[j]$ as 1.
- *committed.* If the proposal $prop_j^e$ is delivered, then the status is *committed* and P_i sets $commit^e[j]$ as 1.

The status is useful for each participant to track the un-delivered proposals and for our framework to achieve its security properties. If the status is *locked*, at least $f+1$ correct participant has already received the proposal. If the status is *finished*, at least $f+1$ correct participants have received the proposal and verified its correctness, which is useful for the supplemental path: a proposal un-delivered in previous epochs can be delivered iff its status is *finished*.

View vector. Each participant maintains a view vector V_i recording the observation on previous proposals. In particular, if P_i has set P_j's proposals as *finished* for all epochs before e, P_i will set $V_i[j]$ as e. At the each of each epoch, participants will agree on $n - f$ view vectors, e.g., $V_{k_1}, V_{k_2}, \ldots, V_{k_{n-f}}$, and compute a common view vector V on them: for $i \in [n]$,

$$V[i] = \max_{f+1}(V_{k_1}[i], V_{k_2}[i], \ldots, V_{k_{n-f}}[i]), \tag{1}$$

where $\max_{f+1}$ represents the $(f+1)$-th largest value. The common view vector V is used to track and commit previously un-delivered proposals.

Now we introduce the details of the Dory framework. The pseudocode is shown in Algorithm 4 and the utility functions are shown in Algorithm 3. The workflow is illustrated in Fig. 5.

Algorithm 4 The Dory framework. Code shown for P_i.

```
    let the $\mathcal{Q}$ of MVBA[$\mathsf{ID}$] be the following predicate:
    $\mathcal{Q}_{\mathsf{ID}}(\{(j_1, h_{j_1}, \sigma_{j_1}), \ldots, (j_{n-f}, h_{j_{n-f}}, \sigma_{j_{n-f}})\}) \equiv$ (for any $k \in [n-f]$, $\mathsf{Verify}_{n-f}(\langle \mathsf{ID}, j_k, h_{j_k} \rangle, \sigma_{j_k}) = 1$)

 1: upon invocation of epoch $e$ do
 2:     Initialization: $lock^e \leftarrow (0_1, \ldots, 0_n)$; $finish^e \leftarrow (0_1, \ldots, 0_n)$; $commit^e \leftarrow (0_1, \ldots, 0_n)$; $S \leftarrow \{\}$; $L_i \leftarrow \{\}$; $In^1 \leftarrow \{\}$; $In^2 \leftarrow \{\}$.
 3:     upon receiving transactions $m_i^e$ to be proposed in epoch $e$ do
 4:         $V_i \leftarrow \mathrm{UpdateView}(e)$                ▷ Broadcast phase
 5:         let $prop_i^e = (m_i^e, V_i)$ be the proposal of epoch $e$
 6:         invoke PB[$\langle e, i \rangle$] with input $prop_i^e$
 7:         upon receiving $(h_i, \sigma_i)$ from PB[$\langle e, i \rangle$] do
 8:             broadcast (LOCK, $e, h_i, \sigma_i$)
 9:     upon receiving $prop_j^e$ from PB[$\langle e, j \rangle$] do
10:         store $prop_j^e$
11:     upon receiving (LOCK, $e, h_j, \sigma_j$) from $P_j$ do
12:         wait until $prop_j^e \neq \perp$
13:             if $\mathcal{H}(prop_j^e) = h_j$ and $\mathsf{Verify}_{n-f}(\langle e, j, h_j \rangle, \sigma_j) = 1$ then
14:                 $lock^e[j] \leftarrow 1$                ▷ Locked
15:                 $\rho_{ji} \leftarrow \mathsf{Sign}_{n-f}(sk_i, \langle e, j, locked \rangle)$
16:                 $L_i \leftarrow L_i \cup \{(j, h_j, \sigma_j)\}$
17:                 send (LOCKED, $e, \rho_i$) to $P_j$
18:     upon receiving (LOCKED, $e, \rho_{ij}$) from $P_j$ do
19:         if $\mathsf{VerifyShare}_{n-f}(\langle e, i, locked \rangle, (j, \rho_{ij})) = 1$ then
20:             $S \leftarrow S \cup \{j, \rho_{ij}\}$
21:             if $|S| = n - f$ then
22:                 $\sigma_i' \leftarrow \mathsf{Combine}_{n-f}(\langle e, i, locked \rangle, S)$
23:                 broadcast (FINISH, $e, \sigma_i'$)
24:     upon receiving (FINISH, $e, \sigma_j'$) from $P_j$ do
25:         if $\mathsf{Verify}_{n-f}(\langle e, j, locked \rangle, \sigma_j') = 1$ then
26:             $finish^e[j] \leftarrow 1$                ▷ Finished
27:     upon $|L_i| = n - f$ then                ▷ Election phase
28:         invoke MVBA[$e$] with input $L_i$
29:     upon receiving $L = \{(j_k, h_{j_k}, \sigma_{j_k})\}_{k \in [n-f]}$ from MVBA[$e$] do
30:         for any $(j_k, h_{j_k}, \sigma_{j_k}) \in L$ do                ▷ Recovery phase
31:             if $prop_{j_k}^e \neq \perp$ and $\mathcal{H}(prop_{j_k}^e) = h_{j_k}$ then
32:                 $lock^e[j_k] \leftarrow 1$                ▷ Locked
33:             $In^1 \leftarrow In^1 \cup \{(e, j_k)\}$
34:         $PP^1 \leftarrow \mathrm{ObtainProposals}(\langle e, 1 \rangle, In^1)$                ▷ 1st AVDD
35:         for any $prop_j^e \in PP^1$ do
36:             extract view vector $V_j$ from $prop_j^e$
37:         $\mathrm{CheckViews}(\{V_j | prop_j^e \in PP^1\}, In^2)$
38:         $PP^2 \leftarrow \mathrm{ObtainProposals}(\langle e, 2 \rangle, In^2)$                ▷ 2nd AVDD
39:         output $\{m_j^e | prop_j^e \in PP^1\} \cup \{m_{j'}^{e'} | prop_{j'}^{e'} \in PP^2\}$
```

Broadcast. The broadcast phase involves n parallel PB instances. At the beginning of each epoch e, each participant P_i first updates V_i by querying the $\mathrm{UpdateView}(e)$ function. The function returns a view vector V_i. $V_i[j]$ stores the latest epoch number, up to which the proposals of P_j are set as *finished* by P_i. Then, P_i includes a batch of transactions m_i^e and V_i as the proposal $prop_i^e$ and starts the i-th PB instance, denoted as PB[$\langle e, i \rangle$]. After PB[$\langle e, i \rangle$] completes, (h_i, σ_i) is returned, where h_i is the hash of $prop_i^e$ and σ_i is a signature for $\langle e, i, h_i \rangle$. Then P_i broadcasts a LOCK message (ln 3-8). Meanwhile, if P_i receives the proposal $prop_j^e$ from P_j in PB[$\langle e, j \rangle$], it stores $prop_j^e$. Additionally, if P_i receives a valid LOCK message from P_j, it creates a signature share for $\langle e, j, locked \rangle$ and sends a LOCKED message to P_j. Finally, each participant that receives $n - f$ signature shares from LOCKED messages will combine the signature shares into a signature σ_i' and then broadcast a (FINISH, e, σ_i') message (ln 9-26). The $lock^e$ and $finish^e$ parameters are updated simultaneously.

Election. After the $n-f$ proposals of epoch e become *locked*,

P_i invokes $\mathrm{MVBA}[e]$ with the *lock* proofs as input (ln 27-28).

Recovery. After $\mathrm{MVBA}[e]$ outputs L, P_i starts the recovery phase. There are two AVDD instances, one for recovering the proposals created in the current epoch, and one for recovering the proposals indexed in the view vectors. In particular, for every $(j_k, h_{j_k}, \sigma_{j_k})$ in L, if P_i has stored $prop_j^e$ but the status is not *locked*, P_i sets the status as *locked*. Then, P_i starts the first AVDD instance by querying the $\mathrm{ObtainProposals}(\langle e, 1\rangle, In^1)$ function. After a vector of proposals PP^1 is obtained from AVDD, P_i extracts the view vectors and combines them into a common vector V, by querying the $\mathrm{CheckViews}(\{V_j | prop_j^e \in PP^1\}, In^2)$ function. Then P_i starts the second AVDD instance to recover the proposals indexed in V by querying the $\mathrm{ObtainProposals}(\langle e, 2\rangle, In^2)$ function. Finally, P_i takes a union of the transactions included in PP^1 and PP^2, and delivers them according to a pre-defined deterministic order (ln 39).

If incorporating Algorithm 4 with Algorithm 1 and Algorithm 2, we instantiate the Dory protocol. In the following, we directly analyze the complexity and correctness of the Dory protocol.

Correctness. We prove the Dory protocol satisfies agreement, total order, liveness and faireness in Sec. VI-B.

Complexity. The Dory protocol achieves $O(1)$ expected time, $O(n^2)$ message and $O(n^2|m| + (\lambda + n)n^2 \log n)$ communication. Moreover, we prove that the concrete communication cost on the payload is only $n^2|m|$ to $3n^2|m|$, which is crucial to the performance, especially when the input is large. Please refer to Sec. VI-B for detailed analysis.

VI. Analysis

In this section, we analyze the correctness and the complexity of the AVDD protocol and the Dory protocol in detail.

A. The AVDD Protocol

Correctness. We prove in the following that Algorithm 1 satisfies the properties defined in Sec. IV-A. The validity property follows from Lemma 2, and the consistency property follows from Lemma 3.

Lemma 1 After broadcasting a Request message, each participant P_i will hold the correct i-th fragments of all its missing elements.

Proof: We assume that P_i broadcasts a Request message carrying an index set R_i. For every $k \in R_i$, P_i sets d_{ki}^* only if it receives consistent fragments through Disperse messages from $f + 1$ different participants, at least one of which is correct. In this way, d_{ki}^* is certainly correct, as no correct participant will send an incorrect fragment. Moreover, since every element in A is held by at least $f+1$ correct participants, P_i can always receive $f + 1$ consistent fragments for every $k \in R_i$. Thus, each participant P_i will hold the correct i-th fragments of all its missing elements. □

Lemma 2 (Validity) At the end of the protocol, every correct participant outputs a complete vector A with no $\perp$ element.

Proof: We assume that P_i broadcasts a Request message carrying an index set $R_i \subseteq [\ell]$. From Lemma 1, each participant P_j will hold either a full data or a j-th fragment for every element in A. Thus, upon receiving a Request message carrying R_i from P_i, P_j will eventually return the j-th fragments of all elements in R_i through Disperse and Confirm messages. Then, for every element requested in R_i, P_i will receive $2f+1$ fragments to trigger the OEC algorithm. Indeed, there may be up to f error fragments from faulty participants. However, P_i will eventually receive $2f+1$ correct fragments from all correct participants, so the OEC algorithm will eventually succeed and output. Therefore, every empty position in vector A_i will be filled. □

Lemma 3 (Consistency) If each participant P_i outputs a complete vector A_i, than every element in A_i is consistent with the one that correct participants hold at the beginning.

Proof: As we assume that every element held by correct participants P_i at the beginning is consistent with that of other correct participants, we only consider its missing elements. For every missing element a_k ready to output, P_i has verified it using $2f+1$ fragments in T_k, of which at least $f+1$ are from correct participants. Due to Lemma 1, every fragment sent by correct participants must be correct. From the nature that $f+1$ points uniquely determine a $(f+1)$-degree polynomial, $f+1$ correct fragments confirm the correctness of a_k. Therefore, every element in A_i is consistent with the one that correct participants hold at the beginning. □

Complexity. It is straightforward that the message complexity of Algorithm 1 is $O(n^2)$ as it only involves all-to-all communication. Now we analyze its communication cost in Lemma 4, which indicates the communication complexity is $O(\ell n|a| + O(\ell n^2 \log n)$.

Lemma 4 The concrete communication cost of the AVDD protocol is bounded by $4\ell n|a| + O(\ell n^2 \log n)$.

Proof: We assume that the number of elements requested by participant P_i is no more than $\ell_i (\ell_i \leq \ell)$ for any $i \in [n]$. Each Request message carries a set of indices of missing elements, which is no more than ℓ_i. Thus, the communication cost of the request phase is at most $n \sum_{i \in [n]} \ell_i$. During the dispersal and confirm phases, each participant P_i receives Disperse and Confirm messages carrying fragments about its requested elements. The size of Reed-Solomon code fragments is $\max(\frac{|a|}{f}, \log n)$ bits. For each participant P_i, the total length of Disperse and Confirm messages received from a participant is at most $2\ell_i(\frac{|a|}{f} + \log n) + O(\ell_i)$ bits. Thus, the communication for P_i in the dispersal and confirm phases is bounded by $n(2\ell_i(\frac{|a|}{f} + \log n) + O(\ell_i))$. Therefore, the total communication cost of the AVDD protocol is at most

$$\begin{aligned} n \sum_{i \in [n]} \ell_i + \sum_{i \in [n]} n(2\ell_i(\frac{|a|}{f} + \log n) + O(\ell_i)) \\ = 2n\frac{|a|}{f} \sum_{i \in [n]} \ell_i + O(n \log n \sum_{i \in [n]} \ell_i). \end{aligned} \tag{2}$$

Moreover, each element in A has been held by at least

$f+1$ correct participants at the beginning, and thus may be requested by at most $2f$ participants, i.e., the total number of requests of all participants is no more than $\ell(2f)$, so we have $\sum_{i\in[n]} \ell_i \le 2\ell f$. Combining it with Equation (2), the communication cost of our AVDD protocol is bounded by $4\ell n|a| + O(\ell n^2 \log n)$. □

B. The Dory Protocol

Correctness. We now prove that the Dory protocol satisfies the definition of Sec. II-C. The agreement property follows from Lemma 7 and Lemma 8. Then, due to the agreement, it is straightforward that Dory achieves total order, as transactions in a single epoch are delivered in the same order according to a pre-defined deterministic algorithm and Dory is invoked sequentially according to monotonically increasing epoch numbers. Liveness follows from Lemma 9. Namely, if some transactions are not delivered in the epoch that they are proposed, they are still able to deliver through the supplemental path. Fairness follows from Lemma 10 as we assume that each participant batches an identical size of transactions as the proposal in each epoch.

Lemma 5 In epoch e, every correct participant will invoke the MVBA instance and get the same output L satisfies that for any tuple $(j, h_j, \sigma_j) \in L$, $\mathsf{Verify}_{n-f}(\langle e, j, h_j\rangle, \sigma_j) = 1$.

Proof: Due to the termination property of PB, all correct participants will complete a PB instance as the sender and broadcast the corresponding *lock* proof. It means that each participant P_i will store $prop_j^e$ and receive valid (h_j, σ_j) from at least $n-f$ correct participants. Thus, P_i will invoke the MVBA instance using a valid L_i as input. Due to the termination of MVBA, after all correct participants invoke the MVBA instance with valid inputs, they will get an output L from it. Then, due to the agreement and external validity of MVBA, the output L that every correct participant gets is the same, and every tuple (j, h_j, σ_j) in L satisfies $\mathsf{Verify}_{n-f}(\langle e, j, h_j\rangle, \sigma_j) = 1$. □

Lemma 6 For any PB instance $\mathrm{PB}[\langle e, k\rangle]$, if any two correct participants P_i and P_j set the status of the corresponding proposal as *locked* and have stored $(prop_k^e)^i$ and $(prop_k^e)^j$ respectively, then $(prop_k^e)^i = (prop_k^e)^j$. Namely, the proposal is correct.

Proof: Suppose $(prop_k^e)^i \ne (prop_k^e)^j$, then P_i and P_j must have received different *lock* proofs, i.e., (h, σ) and (h', σ') where $h = \mathcal{H}((prop_k^e)^i)$ and $h' = \mathcal{H}((prop_k^e)^j)$. It violates the provability property of PB. Thus, P_i and P_j must have stored the same proposal from $\mathrm{PB}[\langle e, k\rangle]$. □

Lemma 7 In each epoch, every correct participant will set In^1 to the same value, and get the same proposals included in PP^1.

Proof: In^1 is determined by MVBA's output L. Due to Lemma 5, every correct participant will get the same L and thus decide the same In^1 by deterministic algorithms. Each proposal indexed in In^1 has been stored by at least $f+1$ correct participants, and these participants will set its status as *locked* because they are all able to see the corresponding *lock* proof due to the agreement of MVBA. By Lemma 6, the AVDD condition for the vector of these proposals is satisfied and every correct participant will get the same PP^1 including them. □

Lemma 8 In each epoch, every correct participant will set In^2 to the same value, and get the same proposals included in PP^2.

Proof: In^2 is determined by the view vectors included in PP^1. Due to Lemma 7, every correct participant will get the same PP^1 and thus decide the same In^2 by deterministic algorithms. In the $\mathrm{CheckViews}$ function (Algorithm 3), the common view vector V is computed by taking the $(f+1)$-th largest value among $n-f$ view vectors for each component in V. Therefore, for any $i \in [n]$, $V[i]$ is no larger than at least one $V_j[i]$ from a correct participant. Namely, for each proposal indexed in In^2, at least one correct participant has set it as *finished*. Thus, at least $f+1$ correct participants have set it as *locked*. Then due to Lemma 6, the AVDD condition for the vector of these proposals is satisfied and every correct participant will get the same PP^2 including them. □

Lemma 9 For any proposal $prop_i^e$ created by a correct participant P_i in epoch e, if it is not delivered in epoch e, then it will eventually be delivered in a later epoch $e'(e' > e)$.

Proof: At the beginning of epoch e, P_i inputs $prop_i^e$ to $\mathrm{PB}[\langle e, k\rangle]$. Due to *Termination* of PB, P_i is able to get the corresponding *lock* proof. Then, since P_i is correct, it will broadcast the *lock* proof to all participants, collect $n-f$ signature shares in LOCKED from correct participants at least, and then broadcast a *finish* proof. Therefore, every correct participant will see the *finish* proof and set $finish^e[i]$ as 1. Later, at the beginning of epoch e', every correct participant will index $prop_j^e$ in the view vector associated with its proposal. In the recovery phase of epoch e', the first AVDD instance will output a vector PP^1 containing $n-f$ view vectors, at least $f+1$ of which are from correct participants. As the common view vector V is computed by taking the $(f+1)$-th largest value among these view vectors for each component of V, $prop_i^e$ must be indexed in V and thus delivered through the second AVDD instance. □

Lemma 10 At any time, at least half of all output proposals are created by correct participants.

Proof: Assuming the most recent completed epoch is e, then there are at most fe faulty proposals in the system as each participant can only propose once in each epoch. According to Algorithm 4, participants will output at least $n-f$ proposals (via the normal path) in each epoch, among which at least $f+1$ proposals are created by correct participants. Therefore, there are at least $(f+1)e$ proposals created by correct participants until epoch e, and it is straightforward that $\frac{(f+1)e}{(f+1)e+fe} > 1/2$.□

Complexity. The Dory protocol achieves $O(1)$ expected time, $O(n^2)$ message and $O(n^2|m| + (\lambda+n)n^2\log n)$ communication. The time complexity clearly is $O(1)$ as dMVBA achieves $O(1)$ expected time and the other protocols we use are deterministic algorithms with a constant number of steps. The message complexity is $O(n^2)$ as the Dory protocol only involves all-to-all communication. The analysis of communication complexity is in Lemma 11. We also prove a concrete communication cost bound on the payload in Lemma 12.

Lemma 11 The communication complexity of the Dory protocol is $O(n^2|m| + (\lambda + n)n^2 \log n)$.

Proof: In the broadcast phase, the input size of each PB instance is $O(|m| + n)$ considering the epoch number is a constant. As the broadcast phase involves n parallel PB instances, the communication complexity is $O(n^2|m| + \lambda n^2 + n^3)$. The election phase has one dMVBA instance, and each participant's input includes $O(n)$ hashes and $O(n)$ signatures. The communication complexity is thus $O(\lambda n^2 \log n)$. We now work on the recovery phase. Recall that in every epoch, any participant will not invoke MVBA until the status of $n - f$ proposals is *locked*. Accordingly, there are at most f proposals that need to be recovered in an AVDD instance of the current epoch or a later one. Due to the communication complexity of AVDD, recovering these missing proposals incurs $O(n^2|m| + n^3 \log n)$ communication. Therefore, the total communication complexity of the Dory protocol is $O(n^2|m|+(\lambda+n)n^2 \log n)$.

Lemma 12 The concrete communication cost on the payload of the Dory protocol is $n^2|m|$ to $3n^2|m|$.

According to Lemma 11, none of the correct participants will request more than f proposals with the same epoch number in AVDD. In particular, until epoch e, the number of requested proposals is up to ef, so the concrete communication cost of the recovery phase is no more than $e(2n\frac{|m|+n}{f} \cdot nf + O(n \log n \cdot nf)) \leq e(2n^2|m| + O(n^3 \log n))$ due to Equation (2) in Lemma 4. Therefore, if we focus on the recovery phase, the concrete communication cost on the payload per epoch is $2n^2|m|$. Adding it with the cost of the broadcast phase, the overall communication cost on the payload is no more than $3n^2|m|$. Moreover, in the optimistic case that all participants have received all proposals in the broadcast phase, the AVDD protocol would not incur any communication. In this case, the communication cost on the payload is only $n^2|m|$, i.e., no expansion.

VII. Evaluation

This section shows our implementation details and evaluation results. We compare Dory with sDumbo in the WAN settings. Our experiments show that 1) Dory significantly saves the communication cost compared with sDumbo, 2) Dory achieves low basic latency—less than 8s even for $n = 151$ participants, 3) Dory achieves high throughput (up to 5$\times$ the throughput of sDumbo), and 4) during failures, Dory exhibits even higher performance than sDumbo (up to 7$\times$ the throughput of sDumbo).

A. Implementation and Experiment Setup

Implementation. We implement Dory and sDumbo in Golang (both open-source[2]) using the same underlying modules, libraries and security parameters. For the network connection, we use TCP sockets to realize reliable point-to-point channels, while running n message sending goroutines and one message receiving goroutine at each participant. For threshold signature and coin-tossing, we use Boldyreva's pairing-based threshold scheme [25] on BN256 curve implemented in kyber[3]; for threshold encryption, we implement Baek and Zheng's scheme [38] on the same curve; for hash function, we use SHA3-512. For Reed-Solomon error correcting code, we use an open-source implementation in infectious[4].

Experiment setup. We evaluate Dory and sDumbo in the WAN settings. Our experiments are deployed on up to 151 Amazon EC2 instances that evenly distributed in up to 10 regions. Each participant runs on a t3.medium instance with two virtual CPUs and 4GB memory. Following the prior works [16], [17], we define the latency as the time interval between the time the first participant starts a new epoch and the time when the $(n-f)$-th correct participant finishes this epoch. We assume that each transaction is a random string of 250 bytes, and participants will input batches of transactions every time. We define the batch size as the number of transactions input by all participants in a single epoch, and varies from 10^2 to 10^6. In all experiments, we run all protocols for ten epochs.

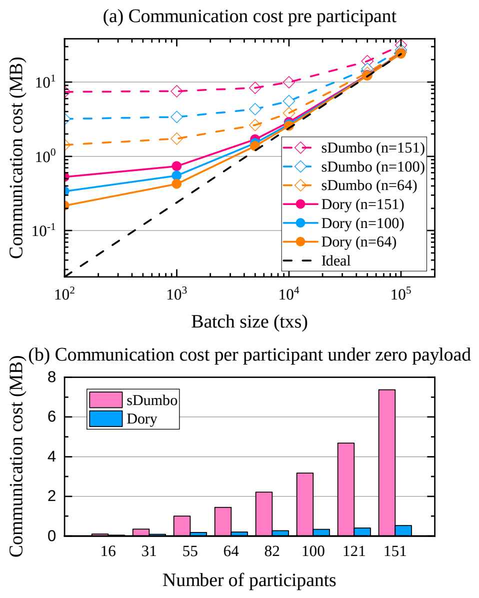


Fig. 8: Communication cost of Dory and sDumbo.

B. Communication Cost

We first evaluate the communication cost of Dory and sDumbo. We measure the total communication bytes for all the messages sent by each participant.

We illustrate the experiment result in Fig. 8(a), where the *ideal cost* is the minimum communication cost (per participant) one could expect for completing the consensus. Dory keeps a tighter distance with the ideal cost than sDumbo when the number of participants scales. For example, when $n = 151$

[2]https://github.com/xygdys/Dory-BFT-Consensus

[3]https://github.com/dedis/kyber

[4]https://github.com/vivint/infectious

TABLE IV: Result highlights of throughput and latency.

Scale	Basic Latency (s)[†]			Peak throughput (tx/s)			Peak throughput with Crash Fault (tx/s)			Peak throughput with Byzantine Fault (tx/s)		
	sDumbo	Dory		sDumbo	Dory		sDumbo	Dory		sDumbo	Dory	
$n = 31$	2.40	1.93	$\downarrow 20\%$	41.1k	131.0k	$3.2\times$	28.0k	43.6k	$1.6\times$	11.5k	53.1k	$4.6\times$
$n = 55$	4.85	2.57	$\downarrow 47\%$	29.1k	120.6k	$4.1\times$	20.2k	41.7k	$2.1\times$	6.6k	39.1k	$5.9\times$
$n = 64$	6.17	3.09	$\downarrow 50\%$	24.9k	117.2k	$4.7\times$	16.2k	41.2k	$2.5\times$	5.9k	38.9k	$6.6\times$
$n = 100$	12.35	4.32	$\downarrow 65\%$	17.4k	87.9k	$5.0\times$	12.2k	32.6k	$2.7\times$	3.5k	24.9k	$7.1\times$
$n = 121$	17.59	5.60	$\downarrow 68\%$	14.5k	77.3k	$5.3\times$	11.2k	31.7k	$2.8\times$	2.6k	19.3k	$7.5\times$

[†]the "Basic Latency" denotes the latency under zero payload, i.e., simply letting each participant input one transaction.

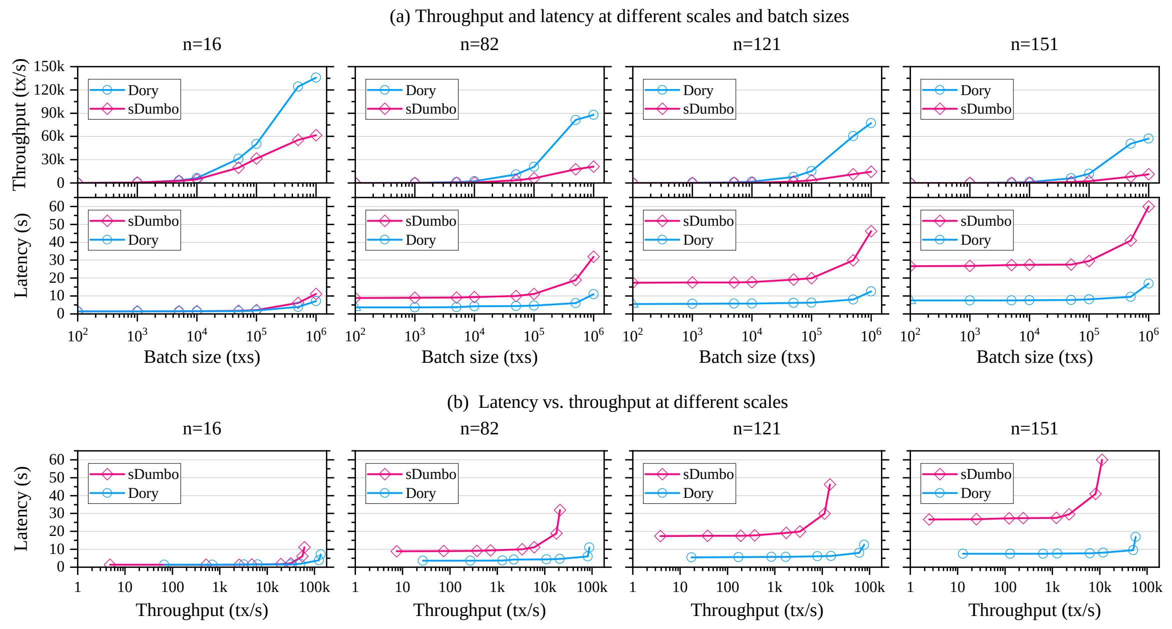


Fig. 9: Throughput and latency of Dory and sDumbo at different participant scales and batch sizes.

and the batch size reaches 10,000, Dory costs about 2.88MB per participant, which is only 21% higher than the ideal, while sDumbo costs about 9.91MB per participant which is $4\times$ that of the ideal cost. Only when the batch size becomes much larger (e.g. $> 10^5$), the communication cost of Dory and sDumbo becomes closer to the ideal cost. This is because the $n^2|m|$ term dominates the communication for large batches.

Zero payload. We also measure the communication cost under zero payload (with only one input transaction for each participant), which evaluates the inherent communication cost of the two protocols and helps understand the following performance difference. As shown in Fig. 8(b), when the number of participants increases from 16 to 151, the communication cost of Dory increases from 45.6KB to just 0.5MB, which is in sharp contrast to that of sDumbo (from 106KB to 7.4MB). This result is consistent with our reduction in the communication complexity.

C. Throughput and Latency

We now show the evaluation results on throughput and latency of Dory and sDumbo. Table IV summarizes some result highlights.

Fig. 9(a) shows throughput and latency of Dory and sDumbo for different network sizes and batch sizes. In terms of both throughput and latency, Dory consistently outperforms sDumbo. In particular, when $n \geq 100$, the throughput of Dory is more than $5\times$ that of sDumbo for *all* batch sizes, and the latency of Dory is significantly lower than that of sDumbo. Moreover, it is straightforward that as the number of participants scales, Dory's performance is higher and more stable compared with sDumbo, which means Dory is more scalable. The reason behind it is that Dory has lower complexity.

We report the latency vs. throughput for different scales in Fig. 9(b), and the peak throughput in Fig. 10(a). Also, for all settings, Dory has shown consistently better performance than sDumbo.

Performance with faults. We also evaluate the performance of Dory and sDumbo with crash fault and Byzantine fault.

For the crash fault experiment, we simply force f participants to crash at all scales. As shown in Fig. 10(b), all protocols suffer a significant reduction in throughput compared with the no-fault scenario, but they offer different fault resilience. Dory is still superior to sDumbo at all scales. An interesting phenomenon is that Dory's advantage over sDumbo is not as significant as in the scenarios with no fault. This is mainly because when we cause f participants to crash, the design to address the wasted proposal issue in the Dory framework becomes ineffective, and both Dory and sDumbo can only output $n - f$ proposals per epoch. However, from another perspective, this also confirms that simply reducing communication complexity can bring nearly

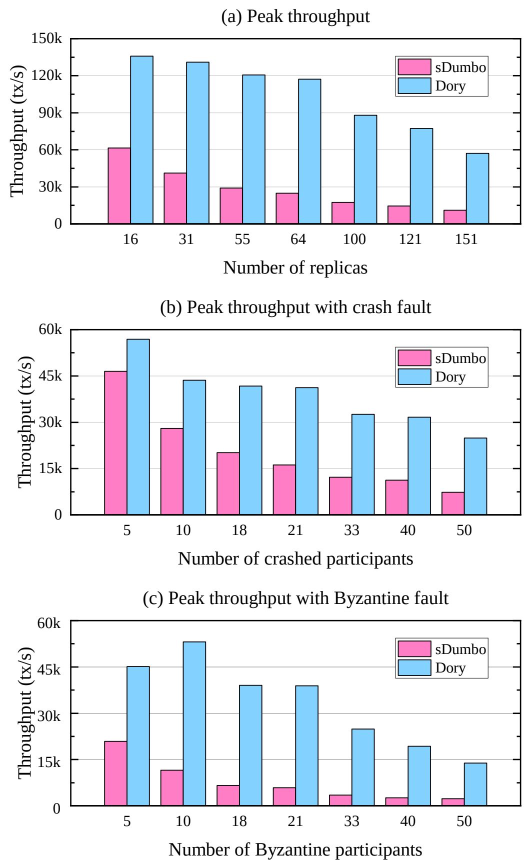


Fig. 10: Peak throughput of Dory and sDumbo with no fault, crash fault, and Byzantine fault.

$3\times$ the throughput improvement.

For the Byzantine fault experiment, we set f malicious participants, each of them sends its proposal only to $f+1$ correct participants in the broadcast phase, and the network is scheduled to make sure the proposals from malicious participants are included in every participant's input for MVBA. In this way, in the recovery phase, at least f correct participants need to reconstruct at least f proposals (created by malicious participants). As illustrated in Fig. 10(c), Dory outperforms sDumbo at all scales. In this experiment, participants would trigger the recovery phase more frequently. We find that Dory has an even more significant advantage over sDumbo compared to that with no fault. This not only shows that Dory has stronger robustness under the malicious attack, but also demonstrates the efficiency of our AVDD protocol.

VIII. Additional Related Work.

Much related work has been discussed in the course of the paper; here we discuss additional related work. The current asynchronous BFT consensus protocols are predominantly based on: asynchronous binary Byzantine agreement (ABA) [13], [16], [33]–[36], multi-valued validated Byzantine agreement (MVBA) [9], [11], [12], [20], and directed acyclic graph (DAG) [21]–[23]. Instantiations derived from the above three paradigms enjoy unique features.

This paper focuses on MVBA-based protocols, which typically have constant time complexity. The first MVBA-based asynchronous BFT consensus protocol is proposed by Cachin et al. [9], but it only proves the theoretical feasibility. After that, the Dumbo family protocols [11], [12], [17] gradually push the MVBA-based asynchronous BFT consensus into practice, among which sDumbo [11] is the state-of-the-art protocol with the same security guarantee as ours. The recent work Dumbo-NG [12] does not achieve the fairness property, and thus it is hard to deploy as the consensus mechanism for permissioned blockchains, where fairness is needed to ensure the blockchain quality [24]. FIN [20] aims to design asynchronous BFT consensus protocols in the signature-free setting.

Compared to our MVBA-based protocol, DAG-based protocols [21]–[23] either have higher complexity or only achieve weak liveness: DAG-Rider [23] requires $O(n^3)$ messages, Tusk [22] also has $O(n)$ communication blowup (using deduplication), and Bullshark [21] aims to introduce garbage collection mechanism into asynchronous BFT consensus protocols and thus sacrifices the liveness.

Another line of work studies ABA-based BFT protocols that do not terminate in constant expected time and have higher message complexity [13], [16], [33]–[36], but they perform well when n is not too large. These protocols can be used to build various applications such as asynchronous MPC [39].

IX. Conclusion

This paper designs and implements Dory, a faster asynchronous BFT consensus protocol with reduced communication compared to existing protocols. We have designed and implemented two building blocks, including a novel primitive called asynchronous vector-data dissemination and a new implementation of multi-valued validated Byzantine agreement. We have also proposed a new asynchronous BFT consensus framework that resolves the wasted proposals issue. We have implemented and deployed Dory and sDumbo using 151 Amazon EC2 instances evenly distributed in 10 regions. We have shown that Dory consistently outperforms sDumbo during both failure and failrue-free scenarios.

References

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

[2] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From byzantine replication to blockchain: Consensus is only the beginning," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 424–436.

[3] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," *Cryptology ePrint Archive*, 2020.

[4] P. Coelho, T. C. Junior, A. Bessani, F. Dotti, and F. Pedone, "Byzantine fault-tolerant atomic multicast," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 39–50.

[5] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 4, pp. 1–39, 2010.

[6] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic, "Xft: Practical fault tolerance beyond crashes." in *OSDI*, 2016, pp. 485–500.

[7] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "Cheapbft: Resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 295–308.

[8] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis *et al.*, "Zeno: Eventually consistent byzantine-fault tolerance." in *NSDI*, vol. 9, 2009, pp. 169–184.

[9] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[10] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, pp. 129–138.

[11] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding dumbo: Pushing asynchronous bft closer to practice," in *Proceedings of the 2022 Network and Distributed System Security Symposium*, 2022.

[12] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2022, p. 1187–1201.

[13] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "Dispersedledger: High-throughput byzantine consensus on variable bandwidth networks," in *NSDI*, 2022, pp. 493–512.

[14] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[15] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *SRDS*. IEEE, 2005, pp. 191–201.

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

[17] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[18] S. Das, Z. Xiang, and L. Ren, "Asynchronous data dissemination and its applications," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2705–2721.

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

[20] S. Duan, X. Wang, and H. Zhang, "Practical signature-free asynchronous common subset in constant time," *Cryptology ePrint Archive*, 2023.

[21] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2705–2718.

[22] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.

[23] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 165–175.

[24] S. Duan, H. Zhang, X. Sui, B. Huang, C. Mu, G. Di, and X. Wang, "Dashing and star: Byzantine fault tolerance using weak certificates," Cryptology ePrint Archive, Paper 2022/625.

[25] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *International Workshop on Public Key Cryptography*. Springer, 2003, pp. 31–46.

[26] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Journal of cryptology*, vol. 17, no. 4, pp. 297–319, 2004.

[27] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in rampart," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 68–80.

[28] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 337–346.

[29] F. J. MacWilliams and N. J. A. Sloane, *The theory of error correcting codes*. Elsevier, 1977, vol. 16.

[30] L. R. Welch and E. R. Berlekamp, "Error correction for algebraic block codes," Dec. 30 1986, uS Patent 4,633,470.

[31] S. Gao, "A new algorithm for decoding reed-solomon codes," in *Communications, information and network security*. Springer, 2003, pp. 55–68.

[32] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[33] M. Ben-Or, R. Canetti, and O. Goldreich, "Asynchronous secure computation," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 52–61.

[34] H. Zhang and S. Duan, "Pace: Fully parallelizable bft from reproposable byzantine agreement," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2022, p. 3151–3164.

[35] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[36] C. Liu, S. Duan, and H. Zhang, "Epic: Efficient asynchronous bft with adaptive security," in *DSN*, 2020.

[37] K. Nayak, L. Ren, E. Shi, N. H. Vaidya, and Z. Xiang, "Improved extension protocols for byzantine broadcast and agreement," in *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[38] J. Baek and Y. Zheng, "Simple and efficient threshold cryptosystem from the gap diffie-hellman group," in *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, vol. 3. IEEE, 2003, pp. 1491–1495.

[39] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 887–903.