Dory: Asynchronous BFT with Reduced Communication and Improved Efficiency

You Zhou
Beihang University
youzhou@buaa.edu.cn

Zongyang Zhang
Beihang University
zongyangzhang@buaa.edu.cn

Haibin Zhang
Beijing Institute of Technology
haibin@bit.edu.cn

Sisi Duan
Tsinghua University
duansisi@tsinghua.edu.cn

Bin Hu
Beihang University
hubin0205@buaa.edu.cn

Licheng Wang
Beijing Institute of Technology
lcwang@bit.edu.cn

Jianwei Liu
Beihang University
liujianwei@buaa.edu.cn

***Abstract*—Asynchronous Byzantine fault-tolerant (BFT) protocols have received increasing attention, as they are particularly robust against timing and performance attacks. This paper designs and implements Dory, an asynchronous BFT protocol with reduced communication and improved efficiency compared to existing systems. In particular, Dory reduces communication both *asymptotically* and *concretely* and gains improved performance. To achieve this goal, we have devised a novel primitive called *asynchronous vector data dissemination*, and moreover, we have developed the technique of *supplemental consensus* originally working with reliable broadcast only, such that the technique can be compatible with the more efficient provable broadcast. We also built Dory-NG by separating data transmission from agreement, just as in Dumbo-NG.**

We have implemented Dory, Dory-NG, Speeding Dumbo (sDumbo), and Dumbo-NG in a new Golang library. Via a deployment using up to 151 replicas on Amazon EC2, we have shown that Dory and Dory-NG consistently outperform sDumbo and Dumbo-NG, respectively—during both failure and failure-free scenarios. For instance, Dory has up to 5x the throughput of sDumbo, while lowering the communication cost for different batch sizes.

I. Introduction

Purely asynchronous Byzantine fault-tolerant state machine replication (BFT) protocols (aka atomic broadcast protocols) assuming no timing assumptions are more robust than the partially synchronous BFT protocols and thus have recently received renewed attention. State-of-the-art asynchronous BFT protocols (with implementations) can be roughly divided into three categories: 1) parallel ABA (asynchronous binary Byzantine agreement) based [1]–[6]; 2) MVBA (multi-valued Byzantine agreement) based [7]–[9]; and 3) DAG (directed acyclic graph) based [10], [11]. All of the instantiations derived from the three paradigms enjoy unique features and are suitable for certain applications.

This paper focuses on the MVBA based protocols, where the state-of-the-art protocols are Speeding Dumbo (sDumbo) [9] and Dumbo-NG [8]. We design and implement Dory with two distinguishing features: 1) reduced communication (both asymptotically and concretely), and 2) improved performance (up to 5× the throughput of sDumbo and 1.7× that of Dumbo-NG).

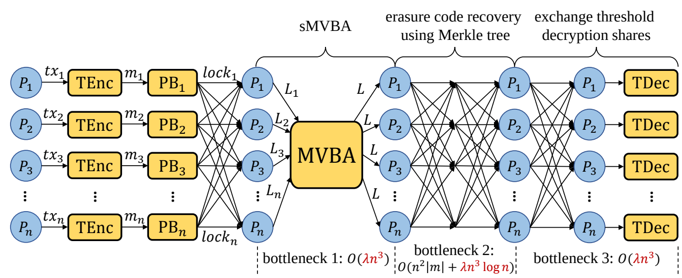

Fig. 1. sDumbo phases and its communication bottleneck. PB stands for provable broadcast.

Overview and communication cost barriers. For MVBA based BFT protocols, the communication bound that one could theoretically hope for (so far) is $O(n^2|m| + \lambda n^2)$ [12], where n is the number of replicas, $|m|$ is the size of proposals and λ is the security parameter. However, known instantiations (as shown in Table I) fall short of the expectation, due to multiple bottleneck components. Take sDumbo [9], a dedicated MVBA based BFT, as an example (Fig. 1). sDumbo has several phases: a PB (provable broadcast) phase sending transactions, a MVBA phase ordering transactions, and a recovery phase obtaining missing transactions. To achieve liveness, sDumbo additionally uses threshold encryption to obfuscate transactions. There are three communication bottleneck ingredients: the MVBA (called sMVBA), the recovery phase, and the threshold decryption phase, incurring $O(\lambda n^3)$, $O(n^2|m| + \lambda n^3 \log n)$, and $O(\lambda n^3)$ communication, respectively. Dumbo-NG [8] separates data transmission from agreement to gain performance improvement compared with sDumbo, but it still suffers similar communication bottlenecks.

Wasted proposals and censorship resilience. Existing MVBA based BFT constructions order proposed transactions from $n - f$ replicas, although in the normal case, all replicas may have proposed and successfully transmitted transactions. Namely, the computation and the bandwidth involving up to f proposals could be "wasted."

A related issue is that if directly allowing replicas to propose transactions in parallel, an adversarial network scheduler can censor transactions in the MVBA phase, thereby attacking censorship resilience (aka liveness). To tackle the issue, one would use:

- threshold encryption [3], which leads to increased computation and λn^3 communication;
- *inter-node-linking* [6] (also called *supplemental consensus* in this paper), which is only possible using more expensive reliable broadcast—resulting in $O(n^3)$ messages;
- pipelined certificates [8], which impacts the blockchain quality—in case one chain led by a faulty replica being extended (unbounded) fast, and sacrifices both $O(1)$ time and state transfer time, as discussed [13].

Our approach. In designing Dory, we have two goals in mind: 1) saving the communication; 2) improving the performance (while maintaining censorship resilience).

First, to push the communication cost closer to the known bound of $O(n^2|m| + \lambda n^2)$ [12], we have carefully designed and implemented the building blocks for Dory, including the first implementation of MVBA achieving $O(n|v| + \lambda n^2 \log n)$ communication, where $|v|$ is the input size of MVBA, and a novel variant of asynchronous data dissemination (ADD) protocol [15] that focuses on the dissemination of *vector-data*, can be triggered only *on request* and reduces the communication concretely (compared to the protocol running ADD in parallel). *Such a primitive is surprisingly challenging to construct; we identified and resolved a subtle dead-lock problem.*

Also, we integrate the technique of supplemental consensus stemming from DispersedLedger [6] in our protocol to achieve *high throughput* as well as *censorship resilience*, while maintaining *low complexity*. Note the supplemental consensus technique works only for reliable broadcast [16], [17] in prior works [6], [8], and the technique requires n reliable broadcast instances incurring $O(n^3)$ messages. *Our key innovation is to enable supplemental consensus using quadratic communication instead of cubic communication.*

A. Our Contributions

We summarize our contributions in the following.

- We propose Dory, a scalable and bandwidth-efficient asynchronous BFT protocol achieving $O(n^2|m| + \lambda n^2 \log n + n^3 \log n)$ communication complexity, edging closer to the known bound [12].
- We propose a new primitive called *asynchronous vector data dissemination* (AVDD) that concretely reduces the communication compared to the one running ADD in parallel.
- We have developed a new technique that simultaneously enables supplemental consensus and censorship resilience in the MVBA framework.
- We implement Dory, Dory-NG (integrating techniques in Dumbo-NG with Dory), sDumbo [9] and Dumbo-NG [8] in Golang, and evaluate them on up to 151 Amazon EC2 instances distributed in 10 regions. Our experimental results demonstrate that Dory has significantly lower communication cost compared with sDumbo, preserves low latency (less than 8s) even for a large network (151 replicas), and achieves high throughput (135k tx/s for 16 replicas and 57k tx/s for 151 replicas)—which is up to 5× the throughput of sDumbo and 1.7× that of Dumbo-NG. We even implemented Dory-NG which consistently outperforms Dumbo-NG too and offers interesting trade-offs to Dory.
- Our library on all the above protocols in Golang has been open-sourced.

II. System and Threat Model

Let $[n]$ denote the set of integers $\{1, 2, ..., n\}$. We consider distributed computing protocols, where f out of n replicas ($\{P_i\}_{i\in[n]}$) may fail arbitrarily (Byzantine failures). The protocols we consider in this work (BFT, MVBA, and AVDD) assume $f \leq \lfloor \frac{n-1}{3} \rfloor$, which is optimal. We consider completely asynchronous systems making no timing assumptions on message processing or transmission delays. We assume reliable channels between replicas. A (Byzantine) *quorum* is a set of $\lceil \frac{n+f+1}{2} \rceil$ replicas. For simplicity, we may assume $n = 3f + 1$ and a quorum size of $2f + 1$. In our protocols, we may associate each protocol instance with a unique session identifier ID, tagging each message in the protocol with ID; we may omit these identifiers when no ambiguity arises. In particular, our BFT protocols proceed in epochs, where in each epoch, replicas agree on the order of a sequence of transactions proposed by replicas.

This paper studies BFT protocols. Our implementations tolerate static corruption, where the adversary needs to choose the set of faulty replicas before the execution of the protocol. We also assume trusted setup for the threshold cryptosystems.

Syntactically, in BFT, a replica *outputs* (atomically deliver) *transactions*, each *being input* by some client. The client computes a final response to its submitted transaction from its responses from replicas. The correctness of a BFT (atomic broadcast) protocol is specified as follows:

- *Agreement.* If any correct replica outputs a transaction m, then every correct replica outputs m.
- *Total order.* If a correct replica outputs a transaction m before outputting m', then no correct replica outputs a transaction m' without first outputting m.
- *Censorship resilience.* If a correct replica inputs a transaction m, then every correct replica eventually outputs m with probability 1.[1]

III. Building Blocks

Multi-valued validated Byzantine agreement (MVBA). MVBA allows each replica that has an input to agree on a value v, which satisfies a global and polynomial-time computable $\mathcal{Q}$ known by all replicas [7]. MVBA is randomized and relies on the coin-tossing protocol which is implemented using unique threshold signature in this paper. More formally, an MVBA protocol satisfies the following properties:

- *Agreement.* If any correct replica outputs v, then every correct replica outputs v.

[1]Due to the celebrated FLP impossibility result [18], asynchronous consensus systems can only be probabilistically live.

TABLE I
COMPARISON FOR PERFORMANCE METRICS

Protocol	Message Complexity	Communication Complexity†	Communication Cost in the Optimistic Case‡	Time Complexity
HoneyBadger [3]	$O(n^3)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$3n^2\|m\| + O(\lambda n^3 \log n)$	$O(\log n)$
DispersedLedger [6]	$O(n^3)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$3n^2\|m\| + O(\lambda n^3 \log n)$	$O(\log n)$
Dumbo [14]	$O(n^3)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$3n^2\|m\| + O(\lambda n^3 \log n)$	$O(1)$
sDumbo [9]	$O(n^2)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$n^2\|m\| + O(\lambda n^3)$	$O(1)$
Dumbo-NG [8]	$O(n^2)$	$O(n^2\|m\| + \lambda n^3 \log n)$	$n^2\|m\| + O(\lambda n^3)$	$O(1)$
Dory (this work)	$O(n^2)$	$O(n^2\|m\| + \lambda n^2 \log n + n^3 \log n)$	$n^2\|m\| + O(\lambda n^2 \log n + n^3)$	$O(1)$

† Communication complexity is measured in bits. λ, the security parameter, denotes the length of signatures and hashes; in practice, λ is no less than 128. (So $O(n^3 \log n)$ is negligible compared to $O(\lambda n^3 \log n)$.)
‡"Optimistic Case" means there is no failure or network asynchrony.

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

Threshold signature. Threshold signature allows any t replicas to produce a valid signature, while any replicas less than t cannot [19], [20]. It consists of the following five algorithms:

- *Key generation:* $\{pk, sk\} \leftarrow \mathsf{KeyGen}(\lambda, n, t)$. Given a security parameter λ, the total number of replicas n and a threshold t, the algorithm outputs a public key pk, and a vector of secret keys $sk = (sk_1, sk_2, \ldots, sk_n)$. 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 if and only if ρ_i is a valid signature share computed by replica P_i for m.
- *Combining:* $\sigma/\perp \leftarrow \mathsf{Combine}_t(m, \{(i, \rho_i)\}_{i \in S})$. Given a set of pairs $\{(i, \rho_i)\}_{i \in S}$, where $S \subset [n]$ and $|S| = t$, the algorithm outputs a signature σ if and only if 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.

Hash. We use a collision-resistant hash function $\mathcal{H}$.

Provable broadcast (PB). PB is a broadcast protocol among n replicas, where a designed replica (also called sender) with ID multicasts some m [7], [9], [21], [22]. Additionally, the sender will also output a tuple (h, σ), where h is the hash of m and σ is a threshold signature for h and ID. Formally, 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. $\mathsf{Verify}_{n-f}(\langle \mathsf{ID}, h \rangle, \sigma) = 1$ and $\mathsf{Verify}_{n-f}(\langle \mathsf{ID}, h' \rangle, \sigma') = 1$, then $h = h'$ and at least $f + 1$ correct replica output m s.t. $\mathcal{H}(m) = h$.
- *Termination.* If the sender is correct and inputs a value m, then all correct replicas will output m. In addition, the sender will output (h, σ) satisfying $\mathcal{H}(m) = h$ and $\mathsf{Verify}_{n-f}(\langle \mathsf{ID}, h \rangle, \sigma) = 1$.

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.

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 $|T| \geq t + 2r$ [23] (e.g., the Berlekamp-Welch algorithm [24], Gao's algorithm [25]).

IV. ASYNCHRONOUS VECTOR DATA DISSEMINATION

As discussed in Fig. 1 in the introduction, one bottleneck is the erasure coding recovery phase which has a $n^2|m| + \lambda n^3 \log n$ overhead. To lower the communication, one intuitive idea is to run the *asynchronous data dissemination* (ADD) protocol proposed recently by Das, Xiang, and Ren [15]. Running ADD-based protocol for n parallel proposals incurs $O(n^2|m| + n^3 \log n)$ communication. However, the ADD protocol is concretely bandwidth-expensive, with a hidden constant of 6 (i.e., $6n^2|m|$).

More critically, the ADD protocol requires replicas to disseminate fragments of the data block to all replicas, no matter whether they need. In sDumbo and Dumbo-NG, the fragments are sent only when replicas do not have the corresponding data blocks. We find that adapting the same idea to parallel ADD is technically challenging, so we define *asynchronous vector data determination* as a first-class primitive in the following.

A. AVDD

Asynchronous vector data dissemination (AVDD). In an AVDD protocol with n replicas, suppose that we have a global ℓ-dimensional vector $M = (m_1, \ldots, m_k, \ldots, m_\ell)$ and for any $k \in [\ell]$, at least $f + 1$ correct replicas hold the same m_k and confirm its correctness. The goal of AVDD is to allow every correct replica to output the common M. Formally, the AVDD

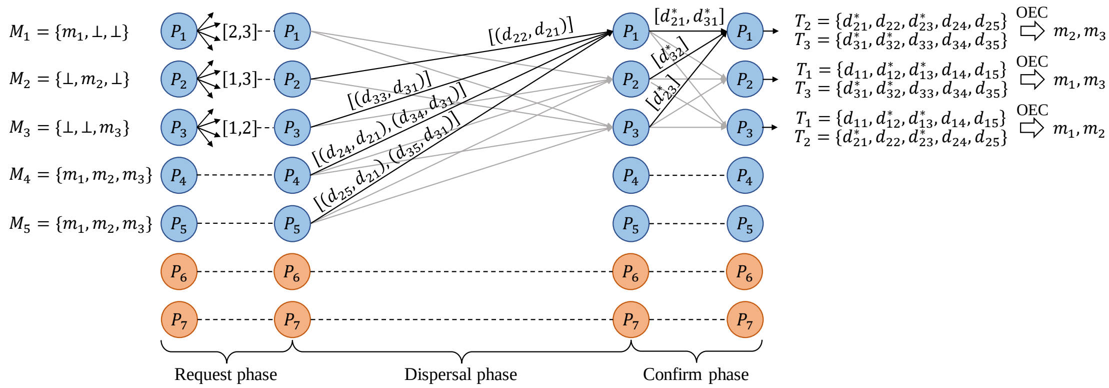

Fig. 2. An example of the AVDD protocol with 7 replicas among which P_6, P_7 are faulty. In the request phase, P_1 requests its missing elements. After the dispersal phase, P_1 will receive $f+1=3$ consistent fragments for m_2, m_3 respectively, and hence set d^*_{21} and d^*_{31}. Similarly, P_2 will 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 m_2 and d^*_{31}, d^*_{32} for m_3. Thus, P_1 is able to collect $2f+1=5$ correct fragments for m_2, m_3 and successfully reconstruct them. Similarly, other replicas also reconstruct their missing elements.

protocol for an ℓ-dimensional vector M satisfies the following correctness property:

- *Correctness.* For any $k \in [\ell]$, if at least $f+1$ correct replicas input the same m_k and other correct replicas input $\bot$, then every correct replica outputs the same vector M.

Overview. Our motivation is to reduce the communication asymptotically and concretely by focusing on the dissemination of *vector-data* instead of ℓ independent data blocks, and meanwhile allowing replicas to exchange the missing data blocks only *on request*. Namely, each replica requests for the missing data blocks and each replica only shares it data blocks upon such requests. As replicas do not need to exchange *all* data blocks, the concrete communication complexity is reduced. However, it is not straightforward to achieve this goal as there might be a dead-lock issue (see Sec IV-B for details).

Our solution provides an elegant way to build vector data dissemination. Roughly speaking, our AVDD protocol is based on the Reed-Solomon error correcting code, and consists of three phases: request, dispersal and an optional confirm phase. In the request phase, every replica will request other replicas to obtain the missing fragments. In the dispersal phase, every replica only sends the fragments it holds (if any) upon receiving a request. For those fragments that the replica does not hold, the replica waits until it receives from other replicas and broadcasts them in the confirm phase. The delay of sending fragments to the confirm phase is crucial for our solution to address the dead-lock issue mentioned above.

The protocol. The pseudocode of the protocol for P_i is shown in Algorithm 1. We now describe it below.

Every replica P_i begins with an input M_i, which consists of a vector of ℓ elements, i.e., $m_1, \ldots, m_\ell$. Depending on the protocol that triggers the AVDD protocol, some elements might be $\bot$. Replica P_i initializes several global parameters: a set S_i tracking the set of elements that need to be reconstructed; a set of values d^*_{ki} for $k \in [\ell]$, where each d^*_{ki} is used to store the data fragments only for the case that $m_k = \bot$; two maps T_i and A_i for storing the received data fragments.

Request. At the beginning of the protocol, P_i first checks M_i and adds k to a set S_i if $m_k = \bot$ for some k such that $1 \le k \le \ell$. If S_i is not empty, P_i broadcasts a $(\textsc{Request}, \mathsf{ID}, S_i)$ message to all replicas (ln 2-8) and then waits for all the

Algorithm 1 AVDD protocol with identifier ID for P_i

1: **Initialization:** $S_i \leftarrow \{\}$; $ReadyFlag \leftarrow false$; **for** $k \in [\ell]$ **do**, $d^*_{ki} \leftarrow \bot$; $T_i \leftarrow \{\}$; $A_i \leftarrow \{\}$
2: **upon** receiving input $M_i = (m_1, m_2, \ldots, m_\ell)$ **do**
3:     **for** $1 \le k \le \ell$ **do**
4:         **if** $m_k = \bot$ **then**
5:             $S_i \leftarrow S_i \cup \{k\}$
6:     **if** S_i is not empty **then**
7:         **broadcast** $(\textsc{Request}, \mathsf{ID}, S_i)$ ▷ Request phase
8:     $ReadyFlag \leftarrow true$

9: **wait** until $ReadyFlag = true$
10:     **upon** receiving $(\textsc{Request}, \mathsf{ID}, S_j)$ from P_j **do**
11:         $S_D \leftarrow \{\}$, $S_C \leftarrow \{\}$
12:         **for** any $k \in S_j$ and $m_k \neq \bot$ **do** ▷ Disperse phase
13:             $(d_{k1}, d_{k2}, \ldots, d_{kn}) \leftarrow \mathsf{Encode}(m_k, n, f+1)$
14:             $S_D \leftarrow S_D \cup (k, d_{ki}, d_{kj})$
15:         **send** $(\textsc{Disperse}, \mathsf{ID}, S_D)$ to P_j
16:         **for** any $k \in S_j$ and $m_k = \bot$ **do**
17:             **wait** until $d^*_{ki} \neq \bot$ ▷ Updated in ln 26
18:             $S_C \leftarrow S_C \cup (k, d^*_{ki})$
19:         **if** S_C is not empty **then**
20:             **send** $(\textsc{Confirm}, \mathsf{ID}, S_C)$ to P_j ▷ Confirm phase
21:     **upon** receiving $(\textsc{Disperse}, \mathsf{ID}, S_D)$ from P_j **do**
22:         **for** any $(k, d_{kj}, d_{ki}) \in S_D$ **do**
23:             $T_i[k] \leftarrow T_i[k] \cup \{(j, d_{kj})\}$
24:             $A_i[k] \leftarrow A_i[k] \cup \{(j, d_{ki})\}$
25:             **if** there are $f+1$ consistent $(\cdot, d_{ki})$ in $A_i[k]$ **then**
26:                 $d^*_{ki} \leftarrow d_{ki}$
27:     **upon** receiving $(\textsc{Confirm}, \mathsf{ID}, S_C)$ from P_j **do**
28:         **for** any $(j, d_{kj}) \in S_C$ **do**
29:             $T_i[k] \leftarrow T_i[k] \cup \{(j, d_{kj})\}$

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

37:     **wait** until no element of M_i is $\bot$
38:         **output** M_i

elements in M_i to become non-empty (ln 37).

Dispersal. If P_i receives an incoming REQUEST message from replica P_j, it checks M_i and initializes two sets, S_D and S_C. S_D is used to store a set of fragments for any $k \in S_j$ and m_k is not $\perp$. S_C is used to store a set of fragments for $k \in S_j$ and m_k is $\perp$. Note that P_i can directly update S_D and send a set of fragments to P_j, but it does not hold any fragments for $m_k = \perp$. In our AVDD protocol, the update of S_C might be deferred but will eventually be completed.

Specifically, we distinguish two cases for each replica P_i:

- Case 1: for any $k \in S_j$ that $m_k \neq \perp$, P_i encodes m_k and obtains the i^{th} and j^{th} data fragments and adds a tuple (k, d_{ki}, d_{kj}) to S_D. After S_D is updated for all such $k \in S_j$, P_i sends a $(\text{DISPERSE}, \mathsf{ID}, S_D)$ message to P_j (ln 12-15). Meanwhile, for the elements P_i requests, upon receiving a $(\text{DISPERSE}, \mathsf{ID}, S_D)$ message from P_j, P_i adds the data fragments to the T_i and A_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 S_i, it will receive at least $f+1$ matching d_{ki} from other replicas (ln 21-26).
- Case 2 (**optional confirm phase**): for any $k \in S_j$ such that $m_k = \perp$, P_i waits until d^*_{ki} is updated in the dispersal phase (case 1) for the messages it requests based on its own S_i. After all such data fragments are updated, P_i adds them to S_C and sends a CONFIRM message to P_j (ln 16-20).

If case 2 is triggered, the replica that requests the corresponding element may receive a CONFIRM message from other replicas. If this is the case, P_i adds the received data fragments to T_i (ln 27-29).

Upon collecting $2f+1$ fragments for any $k \in S_i$, P_i triggers the *online error correcting* (OEC) algorithm [1] to reconstruct m_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 $|T_i[k]| \geq 3f+1$, P_i eventually reconstruct m_k, as mentioned in Section III (ln 30-36).

Finally, P_i waits until it reconstructs all the elements such that there is no $\perp$ in M_i. Then P_i outputs M_i.

B. Discussion, Complexity, and Comparison

Building AVDD in an on-request manner with $O(n^2)$ messages is not easy. Consider the example shown in Fig. 2 where $n = 7$, $f = 2$, and replicas P_6 and P_7 are faulty. P_1 misses m_2 and m_3 and thus requests them via a REQUEST message. Each replica then processes the request from other replicas. A naive approach is that each replica processes each requested element in the REQUEST message one by one, which obviously incurs $O(\ell n^2)$ message complexity, as each replica needs to broadcast up to ℓ messages.

Alternatively, each replica can wait until it has the corresponding data fragments for all the requested elements in a REQUEST message. In this way, the message complexity is $O(n^2)$. Such an approach, however, may have a deadlock issue. For instance, upon receiving a request from P_1, although P_2 holds m_2, it waits until receiving the correct fragments of m_3 from P_3, P_4, and P_5. Further, P_3 will not send the fragment of m_3 to P_2 until receiving the correct fragments of m_1 from P_1, P_4, and P_5. Similarly, P_1 will not send the fragment of m_1 to P_3 until receiving the correct fragments of m_2 from P_2, P_4, and P_5. Accordingly, P_1 waits for P_2, P_2 waits for P_3, and P_3 waits for P_1, creating a deadlock.

TABLE II
COMPARISON BETWEEN AVDD AND ℓ-ADD

Protocol	**Communication Cost**	
	Optimistic Case†	**Worst Case**‡
AVDD	0	$4\ell n\|m\| + O(\ell n^2 \log n)$
ℓ-ADD [15]	$6\ell n\|m\| + O(\ell n^2 \log n)$	

†The "Optimistic Case" means that all replicas are correct and have the complete vector at the beginning.
‡The "Worst Case" means that there are f faulty replicas and each element in the vector is held by only $f+1$ correct replicas at the beginning.

Our AVDD protocol solves it using a dispersal phase and an optional confirm phase, and achieves $O(n^2)$ messages. Moreover, its concrete communication is 0 in the optimistic case (where none of the elements in the M vector is $\perp$ for all replicas), and at most $4\ell n|m| + O(\ell n^2 \log n)$ even in the "worst" case (as we will prove shortly), assuming $|m|$ be the size of each element in the ℓ-dimensional vector M.

In contrast, the data dissemination problem of a ℓ-dimensional vector can be alternatively solved by ℓ parallel ADD [15] instances—hereinafter abbreviated as ℓ-ADD. As shown in Table II, running ℓ-ADD incurs $6\ell n|m| + O(\ell n^2 \log n)$ communication, significantly higher than AVDD.

C. Analysis

We now give an analysis of Algorithm 1. Recall that its goal is to make every correct replica output a common ℓ-dimensional vector $M = (m_1, \ldots, m_k, \ldots, m_\ell)$. We first prove the correctness, i.e, provided that for any $k \in [\ell]$, at least $f+1$ correct replicas input the same m_k and other correct replicas input $\perp$, every correct replica will output the same vector M.

Lemma 1 After broadcasting a REQUEST message, each replica 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 $S_i \subseteq [\ell]$. For every $k \in S_i$, P_i sets d^*_{ki} only if it receives $f+1$ consistent fragments through DISPERSE messages from different replicas, at least one of which is correct. In this case, d^*_{ki} is certainly correct, as no correct replica will send an incorrect fragment. Moreover, since every element in M is held by at least $f+1$ correct replicas, P_i can always receive $f+1$ consistent fragments for every $k \in S_i$. Thus, each replica P_i will hold the correct i^{th} fragments of all its missing elements. ■

Lemma 2 At the end of the protocol, every correct replica outputs the same M. Thus, correctness is satisfied.

Proof: We assume that P_i broadcasts a REQUEST message carrying an index set $S_i \subseteq [\ell]$. From Lemma 1, each replica P_j will hold either a full data or a j^{th} fragment for every element in M. Thus, upon receiving a REQUEST message carrying S_i from P_i, P_j will eventually return the j^{th} fragments of all elements in S_i through DISPERSE and CONFIRM messages. Then, for every element requested in S_i, P_i will receive $2f+1$ fragments to trigger the OEC algorithm. Indeed, there may be up to f error fragments from faulty replicas. However, P_i will eventually receive $2f+1$ correct fragments from all correct replicas, so the OEC algorithm will eventually succeed and output the same element as the one that correct replicas input at the beginning. Therefore, every correct replica will output the same M. ■

Our AVDD protocol has $O(n^2)$ messages as it only involves all-to-all communication. We next analyse its communication cost.

Lemma 3 The concrete communication cost of the AVDD protocol is bounded by $4\ell n|m| + O(\ell n^2 \log n)$.

Proof: We assume that the number of elements requested by replica P_i is no more than $\ell_i (\ell_i \leq \ell)$ for any $i \in [n]$. Then 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 replica P_i receives DISPERSE and CONFIRM messages carrying fragments about its ℓ_i requested elements from n replicas. The size of Reed-Solomon code fragments is $\max(\frac{|m|}{f}, \log n) < \frac{|m|}{f} + \log n$ bits. For each replica P_i and P_j, we consider two situations: (1) P_j holds all P_i's requested elements and sends the fragments through a DISPERSE message, incurring $2\ell_i(\frac{|m|}{f} + \log n) + O(\ell_i)$ bits; (2) P_j holds no P_i's requested elements and sends the fragments through a CONFIRM message, incurring $\ell_i(\frac{|m|}{f} + \log n) + O(\ell_i)$ bits. Thus, the communication for P_i in the dispersal and confirm phases is bounded by $n(2\ell_i(\frac{|m|}{f} + \log n) + O(\ell_i))$. Therefore, the total communication cost of the AVDD protocol is at most

$$\begin{aligned} n \sum_{i\in[n]} \ell_i + \sum_{i\in[n]} n(2\ell_i(\frac{|m|}{f} + \log n) + O(\ell_i)) \\ = 2n\frac{|m|}{f} \sum_{i\in[n]} \ell_i + O(n \log n \sum_{i\in[n]} \ell_i). \end{aligned} \quad (1)$$

Moreover, each element in M has been held by at least $f+1$ correct replicas at the beginning, and thus may be requested by at most $2f$ replicas. In other words, the total number of requests of all replicas is no more than $\ell(2f) = 2\ell f$, i.e.,

$$\sum_{i\in[n]} \ell_i \leq 2\ell f. \quad (2)$$

Therefore, from Equation (1) and Inequality (2), the communication cost of our AVDD protocol is bounded by:

$$\begin{aligned} 2n\frac{|m|}{f} \sum_{i\in[n]} \ell_i + O(n \log n \sum_{i\in[n]} \ell_i) \\ \leq 4\ell n|m| + O(\ell n^2 \log n). \end{aligned} \quad (3)$$

■

Theorem 1 In an asynchronous network of $n = 3f+1$, Algorithm 1 solves the data dissemination problem of a ℓ-dimensional vector with at most $4\ell n|m| + O(\ell n^2 \log n)$ communication.

Proof: From Lemma 2 and Lemma 3, it is immediate that our AVDD protocol satisfies correctness and the communication cost is at most $4\ell n|m| + O(\ell n^2 \log n)$. ■

D. From AVDD to BFT

We can directly use AVDD to build an asynchronous BFT with lower communication compared to existing ones. For instance, as shown in Fig. 3, we can replace the recovery phase of sDumbo with AVDD (setting $\ell = n - f$) to obtain a BFT with lower communication, i.e., $O(n^2|m| + \lambda n^3 + n^3 \log n)$, where the λn^3 term is due to the use of sMVBA and threshold encryption. We will show how to further reduce it in Section V.

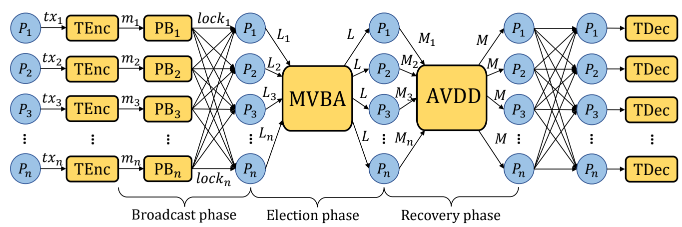

Fig. 3. BFT using AVDD (in sDumbo).

V. THE DORY PROTOCOL

This section presents the design of Dory. We begin with the challenges of reducing the communication and then present our protocol in detail.

A. Building Practical BFT with Lower Communication

The censorship resilience challenge. To cope with the censorship resilience challenge [3], most prior works use a threshold encryption scheme to prevent transactions from being censored by the adversary [4], [9], [14]. The use of threshold encryption scheme incurs a minimum of $O(\lambda n^3)$ communication. In particular, since there are $O(n)$ proposals in each epoch, every replica needs to broadcast $O(n)$ decryption shares of λ size, so the communication is $O(\lambda n^3)$. This term is in general not the bottleneck for prior approaches as the communication of the BFT protocols is higher (e.g., HoneyBadger, BEAT, Dumbo, sDumbo all have $O(n^2|m| + \lambda n^3 \log n)$ communication so the $O(\lambda n^3)$ term is not the bottleneck any more). In this work, we aim to do better and overcome this communication bottleneck.

Using supplemental consensus to enhance the performance. The *supplemental consensus* mechanism, originally used in DispersedLedger [6], provides an efficient approach to utilize the uncommitted but received proposals from prior epochs to enhance the system performance. The core idea is that for the proposals that are not committed in the prior epoch, instead of discarding them directly, replicas can still propose them in the current epoch. Accordingly, replicas reach a supplementary consensus of them while reaching an agreement on the proposals of the current epoch. In addition to enhancing the system performance, this approach naturally solves the censorship resilience issue, as the uncommitted proposals can still be included in the proposals in newer epochs. This approach, however, relies on reliable broadcast protocols (or variants) which incur $O(n^3)$ messages. We show below how security will be violated if we apply it directly to $O(n^2)$ messages framework (i.e. integrated with PB and AVDD).

Integrating supplemental consensus with PB and AVDD. We attempt to integrate supplemental consensus with our proposed approach in Fig. 3 to build a protocol with $O(n^2)$ messages and lower communication—without threshold encryption. The workflow is briefly summarized below.

- First, each replica P_i keeps track of all PB instances and maintains a view vector V_i to keep track of the received proposals. The view vector only stores the instance identifiers instead of the proposals. In particularly, if P_i stores all P_j's proposals up to epoch e, and receives the corresponding $lock$ proofs, P_i will update V_i and set $V_i[j]$ as e.
- At the beginning of an epoch $e'(e' > e)$, each replica P_i includes V_i in its proposal for epoch e'. After the election phase, every replica will decide a common subset consisting of $n-f$ proposals. Every replica first uses an AVDD instance to reconstruct these proposals, where each proposal includes a view vector V_i. Then, based on the $n-f$ view vectors, each replica computes a common view vector V according to Equation (4) shown below. The agreement on the view vectors is called a supplemental consensus.
- Finally, an additional AVDD instance is used to obtain the proposals indexed in V. The union of the transactions from the proposals created in epoch e' and those indexed in V will be delivered.

$$\begin{aligned} V = \{&\max_{f+1}(V_1[1], V_2[1], \dots, V_{n-f}[1]), \\ &\max_{f+1}(V_1[2], V_2[2], \dots, V_{n-f}[2]), \\ &\dots \\ &\max_{f+1}(V_1[n], V_2[n], \dots, V_{n-f}[n])\} \end{aligned} \tag{4}$$

Analysis. Unfortunately, the solution above fails to achieve the agreement property for the proposals indexed in the view vectors. In particular, this is because the common view vector V only includes the instance identifiers and if some proposal m is indexed in V, we can only guarantee that at least one correct replica receives the corresponding $lock$ proof for m. In this case, only one correct replica is able to input m in the additional AVDD instance. However, for AVDD to successfully reconstruct each proposal, we need to guarantee that at least $f+1$ correct replicas input m at the beginning of the AVDD protocol. In fact, the fundamental reason that this solution fails is that the PB primitive cannot achieve the totality property, in contrast to the AVID-M [6] primitive used in DispersedLedger.

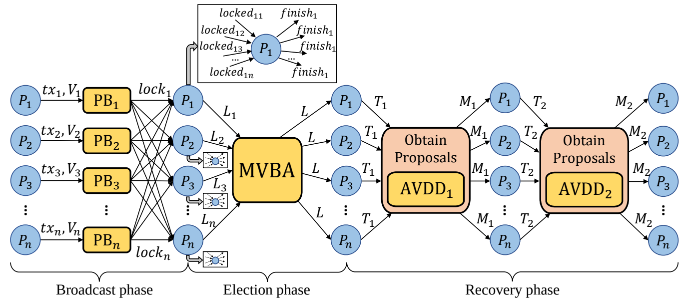

Fig. 4. The workflow of Dory.

In Dory, we use two steps, lock and finish, in addition to each PB instance, to bypass this barrier. It stems from a critical observation that the totality property is unnecessary. To reconstruct some proposal, we only need $n-f$ replicas to hold $lock$ proofs of it, which is ensured by the finish step. The finish step can be executed concurrently with MVBA, as it is useful only for the supplementary consensus.

Moreover, we implement a more efficient MVBA protocol called dMVBA achieving $O(n|v|+\lambda n^2 \log n)$ communication. Based on these changes, we can obtain a secure BFT protocol with $O(n^2|m| + \lambda n^2 \log n + n^3 \log n)$ communication.

B. Dory

We are now ready to present the Dory protocol. As illustrated in Fig. 4, the protocol consists of three phases: broadcast, election, and recovery. Briefly speaking, the workflow of an epoch proceeds as follows. At the beginning, each replica P_i includes its view vector in the proposal. The broadcast phase executes n parallel PB instances where each replica P_i starts an instance to send its proposal to all replicas. After each PB instance completes, the sender P_i broadcasts a $lock$ proof via a LOCK message. After each replica receives at least $n-f$ $lock$ proofs, it provides the $lock$ proofs as input to MVBA and starts the election phase. Meanwhile, to achieve agreement for the supplementary consensus, we need the finish step that can be executed concurrently with MVBA. In particular, upon receiving a $lock$ proof for some proposal, each replica replies with a LOCKED message carrying its signature share to the sender. After receiving $n-f$ valid signature shares, P_i combines the signature shares and sends the signature as a $finish$ proof via a FINISH message. After MVBA outputs, replicas enter the recovery phase that involves two AVDD instances to reconstruct the proposals.

We now present in detail the workflow. According to the progress of the replicas, the status of each proposal m_j^e (created by P_j in epoch e) maintained by replica P_i can be one of the following: $locked$, $finished$, and $committed$, as shown below.

- $locked$. If P_i has received a proposal m_j^e from P_j and receives a $lock$ proof (h, σ) where h is a hash for m_j^e and σ is a valid signature for $\langle e, j, h\rangle$ (i.e., $\mathcal{H}(m_j^e) = h$ and

Algorithm 2 Utility functions of Dory. Code shown for P_i.

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

procedure ObtainProposals(ID, T):
  initialize a $|T|$-dimensional vector M_i, set $k \leftarrow 0$
  for any $(e, j) \in T$ **do**
    if $lock^e[j] = 1$ **then**
      $M_i[k] = m_j^e$
    else
      $M_i[k] = \bot$
    $k \leftarrow k + 1$
  invoke AVDD[ID] with input M_i
  wait until AVDD[ID] outputs M
    for $(e, j) \in T$ **do**
      $lock^e[j] \leftarrow 1$, $finish^e[j] \leftarrow 1$, $commit^e[j] \leftarrow 1$
    return M

procedure CheckViews($views, T$):
  initialize a n-dimensional vector V
  for any $j \in [n]$ **do**
    $V[j] \leftarrow$ the $(f+1)^{th}$ largest value among $\{V_k[j] | V_k \in views\}$
  for any $j \in [n]$ and $1 \le e \le V[j]$ **do**
    if $commit^e[j] = 0$ **then**
      $T \leftarrow T \cup \{(e, j)\}$

Algorithm 3 The Dory protocol. Code shown for P_i.

let the $\mathcal{Q}$ of MVBA[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:** $lock^e \leftarrow (0_1, \ldots, 0_n)$; $finish^e \leftarrow (0_1, \ldots, 0_n)$; $commit^e \leftarrow (0_1, \ldots, 0_n)$; $S_i \leftarrow \{\}$; $L_i \leftarrow \{\}$; $T_1 \leftarrow \{\}$; $T_2 \leftarrow \{\}$.
3:   **upon** receiving transactions tx_i to be proposed in epoch e **do**
4:     $V_i \leftarrow$ UpdateView(e)   ▷ Broadcast phase
5:     **let** $m_i^e = (tx_i, V_i)$ be the proposal of epoch e
6:     **invoke** PB[$\langle e, i \rangle$] with input m_i^e
7:     **upon** receiving (h_i, σ_i) from PB[$\langle e, i \rangle$] **do**
8:       **broadcast** (LOCK, e, h_i, σ_i)
9:   **upon** receiving m_j^e from PB[$\langle e, j \rangle$] **do**
10:     **store** m_j^e
11:   **upon** receiving (LOCK, e, h_j, σ_j) from P_j **do**
12:     **wait** until $m_j^e \neq \bot$
13:       **if** $\mathcal{H}(m_j^e) = h_j$ and $\mathsf{Verify}_{n-f}(\langle e, j, h_j \rangle, \sigma_j) = 1$ **then**
14:         $lock^e[j] \leftarrow 1$ ▷ Locked
15:         $\rho_i \leftarrow \mathsf{Sign}_{n-f}(sk_i, \langle e, j, locked \rangle)$
16:         $L_i \leftarrow L_i \cup \{j, h_j, \sigma_j\}$
17:         **send** (LOCKED, e, ρ_i) to P_j
18:   **upon** receiving (LOCKED, e, ρ_j) from P_j **do**
19:     **if** $\mathsf{VerifyShare}_{n-f}(\langle e, i, locked \rangle, (j, \rho_j)) = 1$ **then**
20:       $S_i \leftarrow S_i \cup \{j, \rho_j\}$
21:       **if** $|S_i| = n - f$ **then**
22:         $\sigma_i' \leftarrow \mathsf{Combine}_{n-f}(\langle e, i, locked \rangle, S_i)$
23:         **broadcast** (FINISH, e, σ_i')
24:   **upon** receiving (FINISH, e, σ_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** $m_{j_k}^e \neq \bot$ and $\mathcal{H}(m_{j_k}^e) = h_{j_k}$ **then**
32:         $lock^e[j_k] \leftarrow 1$ ▷ Locked
33:       $T_1 \leftarrow T_1 \cup \{(e, j_k)\}$
34:     $M_1 \leftarrow$ ObtainProposals($\langle e, 1 \rangle, T_1$)   ▷ 1st AVDD
35:     **for** any $m_j^e \in M_1$ **do**
36:       **decompose** m_j^e into transactions tx_j^e and view V_j
37:     CheckViews($\{V_j | m_j^e \in M_1\}, T_2$)
38:     $M_2 \leftarrow$ ObtainProposals($\langle e, 2 \rangle, T_2$)   ▷ 2nd AVDD
39:     **for** any $m_{j'}^{e'} \in M_2$ **do**
40:       **extract** transactions $tx_{j'}^{e'}$ from $m_{j'}^{e'}$
41:     **output** $\{tx_j^e | m_j^e \in M_1\} \cup \{tx_{j'}^{e'} | m_{j'}^{e'} \in M_2\}$

$\mathsf{Verify}_{n-f}(\langle e, j, h \rangle, \sigma) = 1$), then the status is $locked$ and P_i sets $lock^e[j]$ as 1.

- $finished$. If P_i receives a proof σ' in a FINISH message for the proposal m_j^e (i.e., $\mathsf{Verify}_{n-f}(\langle e, j, locked \rangle, \sigma') = 1$), then the status is $finished$ and P_i sets $finish^e[j]$ as 1.
- $committed$. If the proposal m_j^e is delivered, then the status is $committed$ and P_i sets $commit^e[j]$ as 1.

The status is useful for each replica to track the undelivered proposals and for our protocol to achieve its security properties. If the status is $locked$, at least $f+1$ correct replica has already received the proposal. If the status is $finished$, at least $f + 1$ correct replicas have received the proposal and known its correctness, which is useful for supplementary consensus: A proposal un-delivered in prior epochs can be delivered iff its status is $finished$.

The pseudocode of Dory is shown in Algorithm 3 and the utility functions are shown in Algorithm 2.

Broadcast. The broadcast phase involves n parallel PB instances. At the beginning of each epoch e, each replica P_i first updates V_i by querying the UpdateView(e) function. The function returns a n-dimensional vector V_i, where each component 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 tx_i and V_i as the proposal for the current epoch 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 m_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 m_j^e from P_j in PB[$\langle e, j \rangle$], it stores m_j^e. If P_i receives a valid LOCK message for PB[$\langle e, j \rangle$], it creates a signature share for $\langle e, j, locked \rangle$ and sends a LOCKED message to P_j. Finally, if P_i receives $n-f$ signature shares from the LOCKED messages, it combines the signature shares into a signature σ_i' and then broadcasts a (FINISH, e, σ_i') message (ln 9-26). Each replica P_i keeps track of all the proposals and updates the $lock^e$, $finish^e$, and $commit^e$ parameters according to the description mentioned above.

Election. After the status of $n-f$ proposals of epoch e become $locked$, P_i invokes MVBA[e] providing the $lock$ proofs as input (ln 27-28).

As sMVBA is another communication bottleneck, we implement a more efficient MVBA protocol called dMVBA as shown in Fig. 5. Specifically, we apply the APDB protocol [12] on sMVBA to reduce the communication from $O(n^2|v| + \lambda n^2)$ to $O(n|v| + \lambda n^2 \log n)$ (where $|v|$ is the length of the input for MVBA), while maintaining $O(1)$ time. At the beginning, P_i encodes its input into fragments and disperses them with

Merkle tree witnesses via ECHO messages. Then, upon a receiving valid fragment, each replica 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. Finally, sMVBA will output a tuple (rt_s, σ_s) and replicas will reconstruct the corresponding input as the output of dMVBA. The security of dMVBA follows from [12].

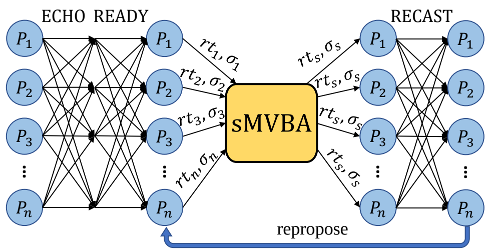

Fig. 5. The workflow of dMVBA.

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 m_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 $\mathrm{ObtainProposals}(\langle e, 1\rangle, T_1)$ function. After a vector of proposals M_1 is obtained from AVDD, each replica obtains the transactions included in the proposals (ln 29-36). Additionally, P_i further extracts the view vectors and combines them into a common vector V, by querying the $\mathrm{CheckViews}(\{V_j | m_j^e \in M_1\}, T_2)$ function. Then P_i starts the second AVDD instance to reconstruct the proposals indexed in V, also by querying the $\mathrm{ObtainProposals}()$ function. A set of transactions are obtained (ln 36-40). Finally, P_i takes a union of the transactions included in the output of two AVDD instances, and delivers them according to a predefined deterministic order (ln 41).

C. Analysis

Lemma 4 In epoch e, every correct replica will invoke the MVBA instance and get some output L from it.

Proof: Due to the termination property of PB, all correct replicas will complete a PB instance as the sender and broadcast the corresponding $lock$ proof. It means that each replica P_i will store m_j^e and receive valid (h_j, σ_j) from at least $n-f$ correct replicas. Thus, P_i will invoke the MVBA instance using a valid L_i as input. Due to the termination of MVBA, after all correct replicas invoke the MVBA instance with valid inputs, they will get an output L from it. ∎

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

Proof: Due to Lemma 4 and the agreement of MVBA, every correct replica will get the same output L. Moreover, due to the external validity of MVBA, 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 replicas P_i and P_j set the status of the corresponding proposal as $locked$ and have stored $(m_k^e)^i$ and $(m_k^e)^j$ respectively, then $(m_k^e)^i = (m_k^e)^j$.

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

By extending Lemma 6, we know that if a replica P_i sets $lock^e[j]$ as 1, then it must have stored the correct m_j^e.

Lemma 7 In each epoch, every correct replica will set T_1 to the same value, and get the same corresponding proposals included in M_1.

Proof: T_1 is determined by MVBA's output L. Due to Lemma 4 and Lemma 5, every correct replica will get the same L and thus decide the same T_1 by deterministic algorithms. Each proposal indexed in T_1 has been stored by at least $f+1$ correct replicas, and these replicas will set its status as $locked$ because they are all able to see the corresponding $lock$ proof due to the agreement of MVBA. Then due to Lemma 6, the AVDD condition for the vector of these proposals is satisfied and every correct replica will get the same M_1 including them. ∎

Lemma 8 In each epoch, every correct replica will set T_2 to the same value, and get the same corresponding proposals included in M_2.

Proof: T_2 is determined by the view vectors included in M_1. Due to Lemma 7, every correct replica will get the same M_1 and thus decide the same T_2 by deterministic algorithms. In the $\mathrm{CheckViews}()$ function (Algorithm 2), 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 replica. Namely, for each proposal indexed in T_2, at least one correct replica has set it as $finished$. Thus, at least $f+1$ correct replicas have set it as $locked$. Then due to Lemma 6, the AVDD condition for the vector of these proposals is satisfied and every correct replica will get the same M_2 including them. ∎

Lemma 9 For any proposal m_i^e created by a correct replica P_i in epoch e, if it is not delivered in epoch e, then it will eventually be delivered in a later epoch e'.

Proof: At the beginning of epoch e, P_i inputs m_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 replicas, collect $n-f$ signature shares in LOCKED from correct replicas at least, and then broadcast a $finish$ proof. Therefore, every correct replica will see the $finish$ proof and set $finish^e[i]$ as 1. Later, at the beginning of some epoch $e'(e' > e)$, every correct replica will index m_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 M_1 containing $n - f$ view vectors, at least $f + 1$ of which are from correct replicas. 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, m_i^e must be indexed in V and thus delivered through the second AVDD instance. ■

Lemma 10 For asymptotic complexity, Dory achieves $O(1)$ expected time, $O(n^2)$ message complexity and $O(n^2|m| + \lambda n^2 \log n + n^3 \log n)$ communication, and costs $n^2|m| + O(\lambda n^2 \log n + n^3)$ communication in the optimistic case.

Proof: The time complexity clearly is $O(1)$ as we follow the classic MVBA-based paradigm: dMVBA achieves $O(1)$ expected time; 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.

We now analyze the communication complexity. In the broadcast phase, the input of each PB instance includes a proposal and a n-dimensional view vector, where the size of transactions is $|m|$ and the size of the view vector is $O(n)$ considering 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 replica'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 replica will not invoke MVBA until the status of $n-f$ proposals is $locked$. Accordingly, none of the correct replicas will request more than f proposals with the same epoch number in any AVDD instance. Therefore, due to Equation (1) in Lemma 3, the communication cost of the recovery phase is no more than $2n\frac{|m|}{f} \cdot nf + O(n \log n \cdot nf) \leq 2n^2|m| + O(n^3 \log n)$, and thus the $O(n^2|m| + n^3 \log n)$ communication complexity.

The optimistic case means there is no failure or network asynchrony. In this case, all replicas will receive all proposals in the broadcast phases, and none of them will trigger the recovery phase. Therefore, the communication cost in the optimistic case of Dory is only $n^2|m| + O(\lambda n^2 \log n + n^3)$.■

Theorem 2 Dory achieves agreement, total order, and censorship resilience with $O(n^2)$ messages, $O(n^2|m| + \lambda n^2 \log n + n^3 \log n)$ communication and $O(1)$ time.

Proof: Agreement follows from Lemma 7 and Lemma 8. Then, due to the agreement, every replica outputs the same transactions in the same epoch. Since the transactions in a single epoch are delivered according to a pre-defined deterministic order and Dory is invoked sequentially according to monotonically increasing epoch numbers, it is straightforward that Dory achieves total order. Censorship resilience 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 consensus. Finally, combining the above conclusions with Lemma 10, Theorem 2 follows. ■

VI. Implementation and Evaluation

We implement[2] Dory, Dory-NG (integrating techniques in Dumbo-NG with Dory), sDumbo, and Dumbo-NG in Golang (both open-sourced), and evaluate them in WAN settings. Our evaluation results show that 1) Dory concretely saves the communication cost compared with sDumbo, 2) Dory achieves low latency—less than 8s even for $n = 151$ replicas, 3) Dory achieves high throughput (135k tx/s for 16 replicas and 57k tx/s for 151 replicas) which is up to 5× that of sDumbo and 1.7× that of Dumbo-NG, and 4) even during failures, Dory and Dory-NG exhibit higher performance than sDumbo and Dumbo-NG.

Implementation. We implement Dory, Dory-NG, sDumbo and Dumbo-NG in Golang using the same underlying modules, libraries and security parameters for a fair comparison. 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 replica. The security parameter λ captures the output length of cryptographic primitives. Specifically, for threshold signature and coin-tossing, we use Boldyreva's pairing-based threshold scheme [19] on BN256 curve implemented in kyber[3], and the length of signatures (or shares) is 64 bytes; for threshold encryption, we implement Baek and Zheng's scheme [26] on the same curve, resulting in decryption shares of 130 bytes; for hash function, we use SHA3-512. For Reed-Solomon error correcting code, we use an open-source implementation in infectious[4] that can easily process transactions at a speed of gigabits per second even for $n = 100$.

Experiment setup. We deploy Dory, Dory-NG, sDumbo and Dumbo-NG on Amazon EC2 using 151 instances where the instances are evenly distributed in up to 10 regions (Singapore, Mumbai, Stockholm, Paris, Frankfurt, St. Paulo, California, Virginia and Canada). Each replica runs on a t3.medium instance with two virtual CPUs and 4GB memory. Following the prior works [3], [14], we define the latency as the time interval between the time the first replica starts a new epoch and the time when the $(n - f)^{th}$ replica finishes this epoch. We assume that each transaction is a random string of 250 bytes which matches the size of basic Bitcoin transactions and replicas will input batches of transactions every time. We define the batch size as the number of transactions input by all replicas in a single epoch (or slot[5] for Dumbo-NG and Dory-NG), and varies from 10^2 to 10^6. Besides, we also evaluate

[2]https://github.com/xygdys/Consensus

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

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

[5]Due to the pipeline and parallel nature of Dumbo-NG and Dory-NG, replicas may propose several times in an epoch, so we define slot as the period of time replicas proposing transactions. Therefore, the batch size of these two protocols represents the number of transactions input by all replicas in a single slot.

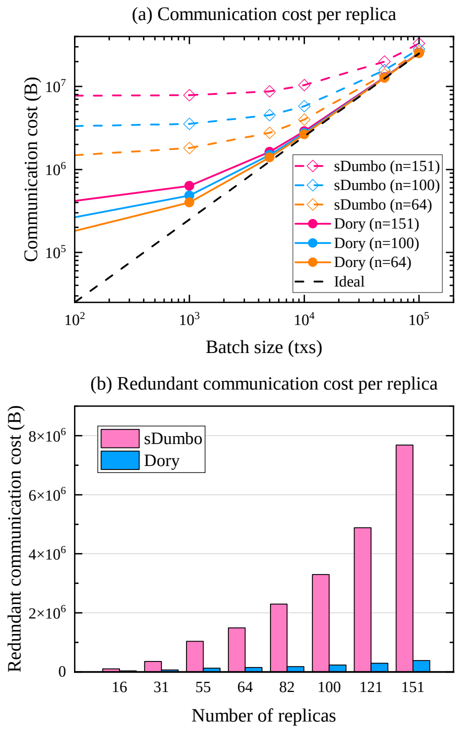

Fig. 6. Communication cost of Dory and sDumbo.

the basic latency which denotes the latency in contention-free scenarios, by simply letting each replica input one transaction, i.e., the batch size is n. In all experiments, we run all protocols for ten epochs and report the average value as the result.

Communication cost. We first evaluate the communication cost of Dory and compare it with sDumbo. We measure the total communication bytes for all the messages sent by each replica while running the protocols. We consider the *ideal cost* as $n|m|$, as this is the minimum communication cost (per replica) one could expect for atomic broadcast; note that the ideal cost also equals the byte length of all transactions in a single batch. We define *redundant communication cost* is the communication cost except for transaction dissemination. As shown in Fig. 6a, though both Dory and sDumbo's communication cost increase as the number of replicas scales, Dory keeps a tighter distance with the ideal cost. For example, when $n = 151$ and the batch size reaches 10,000, Dory costs about 2.75MB per replica, which is only 15% higher than the ideal, while sDumbo costs about 9.91MB per replica 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.

We also visualize the redundant communication cost of the two BFT protocols in Fig. 6b, which helps understand the performance difference between the two protocols. When the number of replicas increases from 16 to 151, the redundant communication cost of Dory increases from 35KB to just 375KB, which is in sharp contrast to that of sDumbo.

Performance. Fig. 7 shows batch size vs. throughput and throughput vs. latency of Dory, sDumbo and Dumbo-NG for different network sizes and batch sizes. For both throughput and latency, Dory consistently outperforms sDumbo. In particular, when $n = 151$, the throughput of Dory is more than $5\times$ that of sDumbo for *all* batch sizes. Dumbo-NG is able to reach high throughput using smaller batch sizes, while its peak throughput is limited. For example, when $n = 121$, the peak throughput of Dumbo-NG is only half of Dumbo-NG's. We report the latency vs. throughput in Fig. 7b. For all settings, Dory has shown consistently better performance than sDumbo and Dumbo-NG.

We report the basic latency for Dory, sDumbo and Dumbo-NG for different network sizes. As shown in Fig. 8, the basic latency of Dory is much lower than that of sDumbo for all the experiments we have conducted. When $n = 151$, the basic latency of Dory is only 7.5s which is only 27% of that of sDumbo. Compared with Dumbo-NG, Dory has almost the same low basic latency while achieving higher throughput and achieve stronger blockchain quality (as mentioned in Sec I).

Dory-NG. While the above experiments have demonstrated that Dory outperforms sDumbo and Dumbo-NG, we still aim at answering the following questions: (1) *can we use Dumbo-NG's technology to further improve Dory*, and (2) *which censorship resilience technology (supplemental consensus or pipelined certificates) is better in performance?* Thus, we implement Dory-NG, by pipelining the broadcast phase and separating it from agreement. As in Dory, Dory-NG uses dMVBA as the randomized engine and uses AVDD to recover proposals.

Fig. 9a shows the peak throughput of Dory, Dory-NG, sDumbo and Dumbo-NG. We find that both Dory and Dory-NG achieve higher peak throughput than sDumbo and Dumbo-NG for all replicas scales. Therefore, we further compare Dory and Dory-NG. Dory-NG outperforms Dory *only* when $n = 16$; When the number of replicas grows, the peak throughput of Dory-NG decreases significantly and is less efficient than Dory. In contrast, Dory has a more stable peak throughput, showing better scalability.

Performance under failures. We also evaluate the performance of Dory, Dory-NG, sDumbo and Dumbo-NG under failure scenarios. For all replicas scales, we force f replicas to crash. As shown in Fig. 9b, all protocols suffer a significant reduction in throughput compared with the failure-free scenario. However, they offer different failure resilience. Dory is superior to sDumbo in all scales, and so is Dory-NG vs. Dumbo-NG. Moreover, Dory-NG and Dumbo-NG outperform Dory when $n \leq 82$, but Dory outpaces Dumbo-NG when $n = 100$ and becomes the fastest when $n \geq 121$. This also confirms the result that Dory is more scalable. In summary, Dory-NG is more suitable for deployment in small-scale and failure scenarios, while Dory can be used in other scenarios.

VII. Additional Related Work

Compared to our MVBA-based protocol, DAG-based protocols [10], [11], [27] have higher communication. In ad-

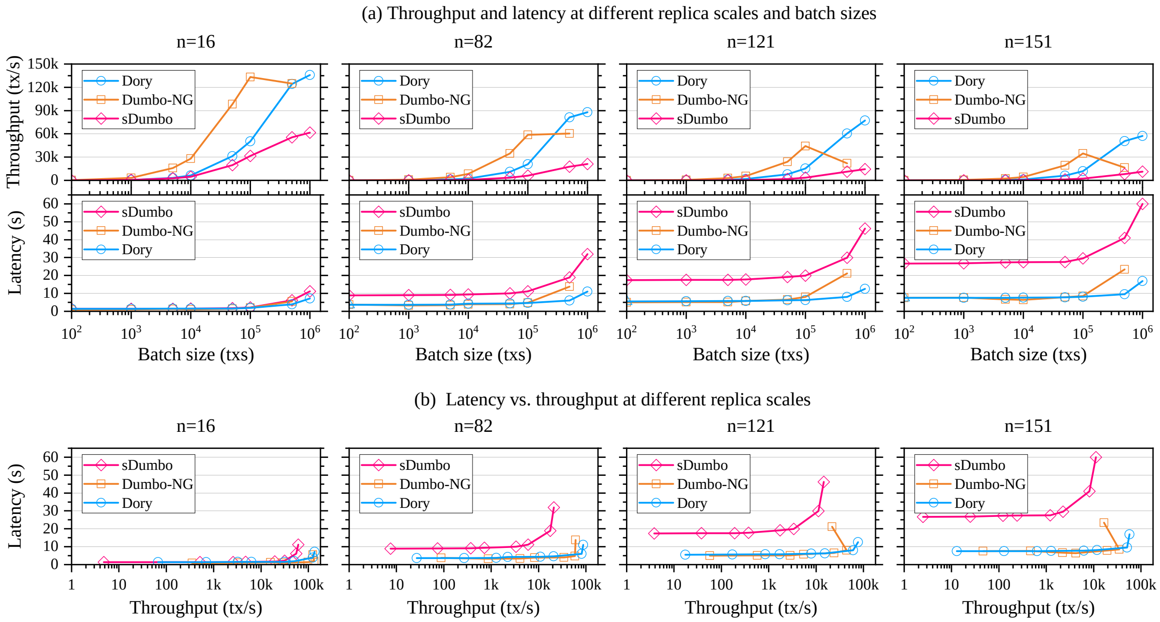

Fig. 7. Throughput and latency of Dory, sDumbo and Dumbo-NG at different replica scales and batch sizes.

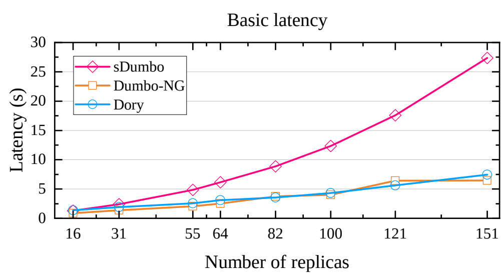

Fig. 8. Basic latency cost of Dory, sDumbo and Dumbo-NG.

dition, DAG-based protocols either achieve weak liveness or suboptimal communication: DAG-Rider requires $O(n^3)$ messages [27], Tusk also has $O(n)$ communication blowup (using de-duplication) [10], and Bullshark achieves weak liveness [11].

Another line of work studies signature-free BFT protocols that do not terminate in constant expected time and have higher message complexity [2]–[5], but they perform well when n is not too large. These protocols can be used to build various other applications such as asynchronous MPC [28].

The MVBA primitive was introduced by Cachin, Kursawe, Petzold, and Shoup [7]. Abraham, Malkhi, and Spiegelman proposed a MVBA protocol [22] that attain $O(n^2|m| + \lambda n^2)$ communication (with optimal word complexity) and additionally achieves a quality property. Lu et al. [12] reduced the communication from $O(n^2|m| + \lambda n^2)$ to $O(n|m| + \lambda n^2)$ by using vector commitments. The recent work from Guo et al. [9] and Gelashvili et al. [29] focused on how to reduce the expected number of rounds while achieving $O(n^2|m| + \lambda n^2)$. Our dMVBA protocol uses the framework of Dumbo-MVBA by Lu et al. [12] but uses sMVBA to save steps. We did

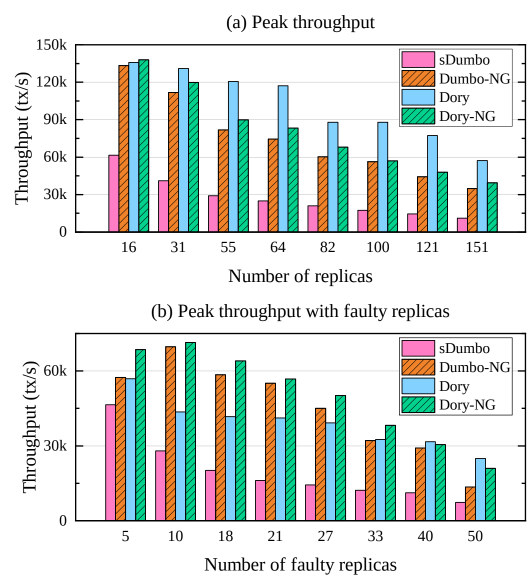

Fig. 9. Peak throughput of Dory, Dory-NG, sDumbo and Dumbo-NG under failure-free and failure scenarios. For Dory and sDumbo, the batch size when reaching their peak throughput is 10^6 in all settings; for Dory-NG, it is 10^5 in all settings; for Dumbo-NG, it is 5×10^5 at 82 replicas under the failure-free scenario, and 10^5 in other settings.

not use pairing-based constant-size vector commitments (e.g., KZG commitments [30]) but chose to use log-size Merkle trees for efficiency; this is also the reason why our dMVBA has $O(n|m| + \lambda n^2 \log n)$ communication (with an additional $\log n$ factor).

VIII. Conclusion

This paper designs and implements Dory and Dory-NG with reduced communication and improved efficiency compared to existing protocols. We designed a novel primitive called asynchronous vector data dissemination, and we developed the idea of supplemental consensus such that it can be compatible with provable broadcast. We have implemented and deployed Dory, Dory-NG, sDumbo, and Dumbo-NG using 151 Amazon EC2 instances evenly distributed in 10 regions. We show that Dory and Dory-NG consistently outperform sDumbo and Dumbo-NG, respectively, during both failure and failrue-free scenarios.

Acknowledgment

The authors thank Lingyue Zhang and Zhuo Wang for their helpful suggestions.

References

[1] 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.

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

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

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

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

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

[26] 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.

[27] 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.

[28] 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.

[29] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 296–315.

[30] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *International conference on the theory and application of cryptology and information security*. Springer, 2010, pp. 177–194.