Byzantine Fault Tolerance with Non-Determinism, Revisited

Yue Huang[1], Huizhong Li[2,3], Yi Sun[2], and Sisi Duan[1]

[1]Tsinghua University
[2]ICT/CAS & UCAS
[3]WeBank Blockchain Team

***Abstract*—Conventional Byzantine fault tolerance (BFT) requires replicated state machines to execute deterministic operations only. In practice, numerous applications and scenarios, especially in the era of blockchains, contain various sources of non-determinism. Meanwhile, it is even sometimes desirable to support non-determinism, and replicas still agree on the execution results. Despite decades of research on BFT, we still lack an efficient and easy-to-deploy solution for BFT with non-determinism—BFT-ND, especially in the asynchronous setting.**

We revisit the problem of BFT-ND and provide a formal and asynchronous treatment of BFT-ND. In particular, we design and implement Block-ND that insightfully separates the task of agreeing on the order of transactions from the task of agreement on the state: Block-ND allows reusing existing BFT implementations; on top of BFT, we reduce the agreement on the state to multivalued Byzantine agreement (MBA), a somewhat neglected primitive by practical systems. Block-ND is completely asynchronous as long as the underlying BFT is asynchronous.

We provide a new MBA construction that is significantly faster than existing MBA constructions. We instantiate Block-ND in both the partially synchronous setting (with PBFT, OSDI 1999) and the purely asynchronous setting (with PACE, CCS 2022). Via a 91-instance WAN deployment on Amazon EC2, we show that Block-ND has only marginal performance degradation compared to conventional BFT.

Keywords*—*BFT, Non-determinism, MBA

I. Introduction

THIS paper revisits the classic problem of Byzantine fault tolerance with non-determinism—BFT-ND. We provide the first practical solution, which is both modular (without the need to modify the consensus layer or the system architecture) and asynchronous (the system being live even during network asynchrony).

Non-determinism in BFT and blockchains. State machine replication (SMR) is a generic approach to achieving system availability and reliability. Byzantine fault-tolerant state machine replication (BFT)—handling Byzantine (arbitrary) failures—is nowadays the de facto model of blockchains [1, 2] and increasingly inspiring the design of permissonless blockchains such as Ethereum [3]. Namely, BFT models the consensus on the order of the transactions, and SMR models the execution of transactions (e.g., smart contracts).

The conventional state machine replication paradigm requires replicated state machines to execute deterministic operations. If all operations are deterministic and replicas execute the operations according to the same order, correct replicas eventually maintain a consistent state. In practice, various scenarios contain non-determinism—caused by, for instance, scheduler decisions, multi-threading and parallel execution, probabilistic algorithms, operating system discrepancy, and implementation differences. Namely, even if all correct replicas execute the transactions in the same logical order, they end up with inconsistent system states.

Take the programming languages in blockchain smart contracts as examples. The Chaincode in Hyperledger Fabric [1] uses general-purpose languages and naturally contains non-deterministic operations (due to, e.g., local random numbers [4]). While the programming languages of Ethereum virtual machine (EVM) do not permit non-deterministic operations [5], Ethereum still suffers from various inconsistencies of execution results because of, for example, the discrepancy of the virtual machine versions and programming languages [6]. While some engineering efforts can minimize state diverging (e.g., by disabling non-deterministic operations or making contract execution publicly auditable), it is still difficult to detect and quarantine all possible sources of non-determinism.

In fact, non-deterministic operations might sometimes be *desirable* in blockchains. For example, one issue for Ethereum and its successors (due to the use of EVM) is that they "do not support non-deterministic operations and cannot securely operate on private data." [7] Many prior systems also pointed out that building a system that handles non-determinism by design is useful in some applications [8]–[13]. For example, multi-threaded execution of the transactions, typically designed for enhanced performance, is also one source of non-determinism. In this case, replicas still need to agree on the result, and the underlying BFT-SMR should be designed to support non-determinism [10].

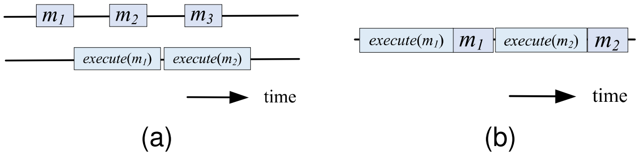


Figure 1. Models of dealing with non-determinism in BFT. (a) Order-then-execute. (b) Execute-then-order

Cachin, Schubert, and Vukolić (CSV) [8] provide a comprehensive survey on protocols dealing with non-determinism in BFT and distinguish three models:

- *Order-then-execute (Figure 1a).* The transactions are first ordered using BFT and then executed at replicas; the executed results, one from each replica, are communicated to all other replicas using (up to) n BFT instances (where n is the number of replicas). Then, a decision can be made based on the atomically delivered outputs.
- *Execute-then-order (Figure 1b).* All replicas execute the transactions speculatively upon receiving requests from a designated replica (i.e., the leader). Then, the leader collects signed approvals from replicas. After receiving $f + 1$ approvals for the same speculative result, the leader initiates a BFT protocol communicating the decision and the signed approvals to all other replicas.
- *Primary-backup.* A specific replica is assigned as the primary, making all non-deterministic choices, while other replicas act as backups and follow the choices. In the Byzantine failure scenario, the primary must provide the state and the correctness proof of the execution to justify the choices and the results.

In particular, the order-then-execute and the execute-then-order approaches do not require modifying the source code of the BFT protocol; however, the primary-backup approach requires that the developers have access to and modify the protocol. CSV proposed an approach in the execute-then-order model, a variant of which is used in Hyperledger Fabric [14].

Issues. In spite of decades of research on BFT, we still lack an efficient and easy-to-deploy solution for BFT-ND. In particular, existing approaches suffer from the following issues:

- *Modularity and compatibility.* So far, existing solutions dealing with non-determinism still lack modularity, in the sense that they need to 1) modify the underlying BFT protocol—in which case a special-purpose BFT protocol handling non-determinism should be designed, validated, and implemented, and/or 2) modify the system architecture (e.g., by adopting the execute-then-order model).
  First, designing and implementing a new BFT protocol (especially at the production level) has been acknowledged as a challenging task; crucially, for BFT infrastructures in operation, transitioning to new ecosystems would be prohibitively expensive. Second, it is not always possible to change the system architecture as it may take tremendous engineering efforts.
- *Efficiency.* Existing BFT-ND protocols are much less efficient than conventional BFT protocols. First, the order-then-execute approach allows replicas to reach an agreement on the order of transactions and then execute the transactions. If non-deterministic operations are detected, replicas roll back to the previous state until an agreement on the state is reached. When unwanted rollback is triggered frequently, the entire system may suffer from significant performance degradation. Second, the execute-then-order and primary-backup approaches require replicas to execute the transactions first and then reach an agreement on the execution results. As the execution of the transactions and the agreement are highly coupled, the *slower* process becomes the bottleneck of the system.
- *Asynchrony.* Existing BFT protocols with non-determinism do not have effective solutions dealing with network asynchrony [8]–[13]. First, the order-then-execute paradigm would require n-fold increase in message and communication to cope with asynchrony. The other two approaches inherently rely on a leader to prevent the replica state from diverging; it is unclear how to extend them to deal with network asynchrony.

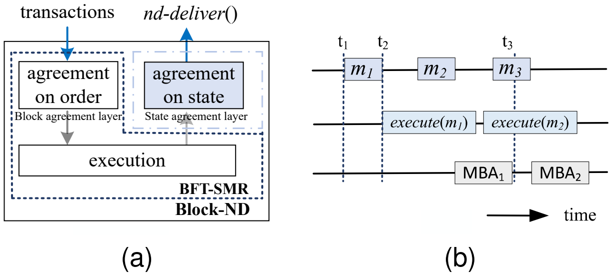


Figure 2. Overview of Block-ND. (a) Block-ND architecture. (b) Block-ND workflow

Our approach. The root cause for all the challenges above is that traditional BFT-ND protocols handle the agreement on the order of transactions and the agreement on the replica state at the same time. In this paper, we challenge this conventional wisdom and separate it into two tasks: agreement on the order of the transactions (block agreement layer) and agreement on the state (state agreement layer). In particular, we design Block-ND, the architecture of which is shown in Figure 2a. The block agreement layer is fully de-coupled from the state agreement layer; the agreement on the state additionally allows the replica state to converge. The block agreement layer employs a conventional BFT protocol, allowing one to reuse the existing BFT system that has been implemented and deployed.

We reduce the problem of agreement on the state to multivalued Byzantine agreement (MBA), a primitive that allows correct replicas to reach an agreement on some arbitrary values. MBA guarantees that replicas eventually agree on the state by some correct replica(s). To further capture the need for state transfer and make the agreement on the state more self-contained, we slightly extend the MBA notion to a new primitive called double-output multivalued Byzantine agreement (DO-MBA). DO-MBA produces two outputs: the primary output follows that of a conventional MBA; the secondary output denotes whether a replica needs to synchronize its state with other replicas. In this way, DO-MBA fully captures our needs for agreement on the state and ensures replicas eventually converge on their states.

As shown in Figure 2b, Block-ND follows the order-then-execute paradigm. Replicas first reach an agreement on the order of blocks of transactions (e.g., at time t_2, the order of m_1 is committed), execute the transactions in the background, and then start an MBA instance to reach an agreement on the execution results. Replicas can continue the block agreement layer without waiting for the execution and MBA to complete. Our approach is completely asynchronous: if the underlying BFT protocol is asynchronous, Block-ND is asynchronous; if the underlying BFT is partially synchronous [15], the mechanisms ensuring liveness are "hidden" by BFT itself.

Our contributions. We make the following contributions:

- We revisit the problem of BFT-ND. We separate the agreement on the order of transactions and the agreement on the state. To reach an agreement on the state, we reduce the problem to multivalued Byzantine agreement (MBA).
- We present a practical and asynchronous MBA construction ND-MBA based on reproposable asynchronous binary agreement (RABA) [16]. Our MBA protocol terminates in only three steps in the optimistic case and is more efficient than all MBA constructions we are aware of. Accordingly, the agreement on the state can be very lightweight when all correct replicas hold the same state, i.e., there are no non-deterministic operations.
- We transform MBA construction to DO-MBA, an extended primitive of MBA that has two outputs. In our case, transforming ND-MBA to DO-MBA is easy: we only need to modify a few lines of code. DO-MBA might be a primitive of independent interest.
- Based on the MBA protocol, we build Block-ND. Block-ND can reuse any BFT protocols and is asynchronous as long as the underlying BFT is asynchronous.
- We evaluate the throughput and latency of our DO-MBA protocol and Block-ND using up to 91 Amazon EC2 instances. We provide a partially synchronous instantiation and an asynchronous instantiation using PBFT [17] and PACE [16], respectively. Our results show that Block-ND is efficient, with 0.89%-21.0% performance degradation compared to the underlying BFT protocols that do not handle non-determinism.

II. Related Work

BFT assuming partial synchrony and asynchrony. BFT protocols can be divided into partially synchronous protocols (e.g., [17]–[20]) and asynchronous protocols (e.g., [16, 21]–[24]). Partially synchronous BFT assumes that there exists an unknown upper bound on the message transmission and processing delay [15]. In contrast, asynchronous BFT assumes no timing assumptions. The celebrated FLP result [25] rules out the possibility of deterministic consensus in the asynchronous environment; asynchronous BFT must thus be randomized to be probabilistically live. Block-ND, by design, assumes no timing assumptions, applying to both partially synchronous and asynchronous BFT protocols.

Detection of non-determinism in blockchains. A line of work aims to *detect* non-deterministic behavior via code analysis (see [26] and references therein). For example, Luu et al. used static code analysis to study non-determinism on transaction dependencies in EVMs during the ordering and execution of transactions [27]. Some commercial software tools can also analyze deployed contracts and detect non-deterministic operations [28]. These analysis-based approaches, however, can only detect *program-level* non-determinism with limited accuracy.

Separating agreement from execution. Yin et al. [29] and Duan et al. [13] studied the architecture of separating the BFT agreement and the execution of transactions. In the architecture, the BFT agreement cluster orders client requests and the execution cluster then executes client requests according to the order. Our work is different from their approaches: we first reach an agreement on the transactions from different clients and then reach an agreement on states across correct replicas, even in the presence of non-determinism.

Multivalued Byzantine agreement (MBA). In synchronous settings, the reduction from MBA to BA was first introduced by Turpin and Coan [30] and followed up by [31]–[33]. In the asynchronous environments, the reduction from MBA to asynchronous BA (ABA) was first established by Correia, Neves, and Veríssimo [34]. The MBA protocol, however, has $O(n^3)$ message complexity and expected $O(1)$ time. Mostéfaoui and Raynal [35] presented the first MBA with optimal $O(n^2)$ message complexity and optimal expected constant time. In this work, we build a new MBA protocol that significantly reduces the number of communication steps of prior protocols while maintaining the optimal message and time complexity.

RABA. Reproposable ABA (RABA) was originally proposed by Zhang and Duan [16]. Unlike prior RABA-based distributed computing primitives [16,24,36,37], the usage of RABA in our ND-MBA protocol is radically different. Indeed, all previous RABA-based approaches are used to build asynchronous common subset (ACS) [16, 24, 36] and distributed key generation (DKG) [37]. In contrast, our work uses RABA to build MBA.

III. System Model

We consider a system with n replicas, where f of them may be Byzantine (fail arbitrarily). The protocols we consider assume $f \leq \lfloor \frac{n-1}{3} \rfloor$, which is optimal. According to the timing assumptions, BFT protocols can be divided into partially synchronous protocols (where messages are guaranteed to be delivered within an unknown time-bound [15]) and asynchronous protocols (with no timing assumption). Partially synchronous BFT attains liveness only when the network becomes synchronous. Asynchronous BFT can (always) use randomization to achieve probabilistic liveness.

In our description, we tag a protocol instance with a unique identifier id. We may omit the identifiers.

Throughout the paper, we explicitly distinguish between BFT (atomic broadcast), BFT-SMR (secure with deterministic operations only), and BFT-ND (dealing with non-determinism, to be defined in Sec. V).

BFT (atomic broadcast). This paper uses BFT and atomic broadcast interchangeably, as these two primitives are only syntactically different. In atomic broadcast, a replica *a-broadcasts* messages and all replicas *a-deliver* messages. The correctness of atomic broadcast is specified as follows:

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

The atomic broadcast abstraction implicitly assigns an order to each delivered transaction. Slightly restricting its syntax, we may write *a-deliver*(sn, m) to denote that m is the sn-th *a-delivered* transaction.

SMR and BFT-SMR. In the *state machine replication* paradigm [38], a state machine consists of a set of states $\mathcal{S}$, a set of operations $\mathcal{O}$, and an execution function *execute*(). The execution function *execute*() takes a state s (initially s_0) and an operation o as input and outputs an updated state s': *execute*$(s, o) \to s'$. A state machine can (optionally) compute a response r based on its state. Alternatively, one could also include a response in the output of the execution function: *execute*$(s, o) \to (s', r)$.

In the BFT-SMR protocol, each replica maintains a replicated state machine, and all correct replicas maintain the same initial state. If they use atomic broadcast (BFT) to disseminate and order client operations, then once the operations are deterministic, their states will never diverge. Namely, atomic broadcast directly implies a secure BFT-SMR for deterministic operations.

IV. Preliminaries

Asynchronous binary agreement (ABA). An ABA protocol can be viewed as a binary version of MBA with the input domain being $\{0, 1\}$. An ABA protocol is specified by two events: *a-propose*() and *a-decide*(). Every replicas *a-proposes* a bit $v \in \{0, 1\}$, and each correct replica *a-decides* a value $v \in \{0, 1\}$. ABA should satisfy the following properties:

- **Validity**: If all correct replicas *a-propose* v, then any correct replica that terminates *a-decide* v.
- **Agreement**: If a correct replica *a-decides* v, then any correct replica that terminates *a-decides* v.
- **Termination**: Every correct replica eventually *a-decides* some value.
- **Integrity**: No correct replica *a-decides* twice.

Reproposable asynchronous binary agreement (RABA). RABA is a distributed computing primitive recently introduced by Zhang and Duan [16]. In contrast to conventional ABA protocols, where replicas can vote once only, RABA allows replicas to change their votes. A RABA protocol is specified by *r-propose*(), *r-repropose*(), and *r-decide*(), with the input domain being $\{0, 1\}$. For our purpose, RABA is "biased towards 1." Each replica *r-proposes* a value b at the beginning of the protocol. A correct replica that *r-proposed* 0 is allowed to change its mind and *r-repropose* 1, but not vice versa. If a replica *r-reproposes* 1, it does so at most once. RABA (biased towards 1) satisfies the following properties:

- **Validity**: If all correct replicas *r-propose* v and never *r-repropose* $\bar{v}$, then any correct replica that terminates *r-decide* v.
- **Unanimous termination**: If all correct replicas *r-propose* v and never *r-repropose* $\bar{v}$, then all correct replicas eventually terminate.
- **Agreement**: If a correct replica *r-decides* v, then any correct replica that terminates *r-decides* v.
- **Biased validity**: If $f + 1$ correct replicas *r-propose* 1, then any correct replica that terminates *r-decides* 1.
- **Biased termination**: Let Q be the set of correct replicas. Let Q_1 be the set of correct replicas that *r-propose* 1 and never *r-repropose* 0. Let Q_2 be correct replicas that *r-propose* 0 and later *r-repropose* 1. If $Q_2 \neq \emptyset$ and $Q = Q_1 \cup Q_2$, then each correct replica eventually terminates.
- **Integrity**: No correct replica *r-decides* twice.

Multivalued Byzantine agreement (MBA). An MBA protocol is specified by two events: *mba-propose*() and *mba-decide*(). Every replica *mba-proposes* an input value $v_i \in \{0, 1\}^L$, and each correct replica *mba-decides* an output $v \in \{0, 1\}^L$, where L is a finite integer. Let $\perp$ be a distinguished symbol. An MBA protocol should satisfy the following properties.

- **Validity**: If all correct replicas *mba-propose* v, then any correct replica that terminates *mba-decides* v.
- **Agreement**: If a correct replica *mba-decides* v, then any correct replica that terminates *mba-decides* v.
- **Termination**: If all correct replicas *mba-propose* some value, every correct replica eventually *mba-decides*.
- **Integrity**: No correct replica *mba-decides* twice.

Note that the following non-intrusion is an optional property that can be met in some MBA constructions only [35]:

- **Non-intrusion**: If a correct replica *mba-decides* v such that $v \neq \perp$, then at least one correct replica *mba-proposes* v.

Crusader agreement (CA). CA [39] relaxes the notion of Byzantine agreement (MBA and binary agreement). In CA, it is allowed that some correct replicas decide a $\perp$ value, while other correct replicas decide the same non-$\perp$ value. A CA protocol is specified by *c-propose*() and *c-decide*() and satisfies the following properties.

- **Weak agreement**: If a correct replica *c-decides* value v and another correct replica *c-decides* v', then $v = v'$ or one of v and v' is $\perp$.
- **Validity**: If all correct replica *c-propose* v, then any correct replica that terminates *c-decides* v.
- **Termination**: If all correct replicas *c-propose* some value, every correct replica eventually *c-decides*.

Hash; threshold signatures. We use a collision-resistant hash function $hash()$. We also use threshold signatures in our DO-MBA protocol. A (ℓ, n) threshold signature scheme [40, 41] consists of the five algorithms (*tgen*, *tsign*, *shareverify*, *tcombine*, *tverfiy*). *tgen* outputs a public key and a vector of n verification keys $\boldsymbol{vk}$ known to anyone and a vector of n private keys. A partial signature signing algorithm *tsign* takes as input a message m and a private key sk_i, and outputs a partial signature π_i. A share verification algorithm *shareverify* takes input pk, verification key vk_i, a message m, and a partial signature π_i, and outputs a bit. A combining algorithm *tcombine* takes as input pk, the verification keys $\boldsymbol{vk}$, a message m, and a set of ℓ valid partial signatures, and outputs a signature π. A signature verification algorithm *tverify* takes as input pk, a message m, and a signature π, and outputs a bit. We require the conventional robustness and unforgeability properties for threshold signatures.

Convention and notation. In the paper, we use *best-effort* *broadcast*, or simply *broadcast*, where a sender multicasts a message to all replicas. To measure the latency of asynchronous protocols, we use the standard notion of *asynchronous steps* [42], where a protocol runs in x asynchronous steps if its running time is at most x times the maximum message delay between correct replicas during the execution. The notion of *rounds* is restricted to ABA protocols: an ABA protocol proceeds in rounds, where an ABA round consists of a fixed number of steps.

V. Pathway to Block-ND

A. Formalizing BFT-SMR with non-determinism (BFT-ND)

If allowing sources of non-determinism, we need to carefully revisit the properties of BFT-SMR. In this case, atomic broadcast (interchangeable with BFT) does not imply a "secure" BFT-SMR; indeed, the states of replicas may diverge due to non-determinism.

We, therefore, define BFT-SMR with non-determinism, or BFT-ND. Still, in BFT-ND, we use the same syntax as SMR. A client still submits a transaction containing some operation o and may expect a response r from the replicas. However, we dissociate the events in BFT-ND from those in the atomic broadcast protocol. Namely, we do not consider the *a-broadcast* and *a-deliver* events. Instead, we define *nd-deliver*(o) as the event that a replica terminates the BFT-ND protocol and updates its state via an *update* function based on input o. (Each replica may internally run *a-broadcast*, *a-deliver*, *execute*$()$, and possibly other operations, but these functions need not be exposed as the API of BFT-ND.) Specifically, we consider the following properties for BFT-ND:

- **Total order**: If a correct replica *nd-delivers* o before *nd-delivering* o', then no correct replica *nd-delivers* o' without first *nd-delivering* o.
- **Correctness**: If a correct replica maintains state s before it *nd-delivers* o and maintains s' after it *nd-delivers* o, another correct replica maintains state s before it *nd-delivers* o and maintains s'' after it *nd-delivers* o, then $s' = s''$.
- **Liveness**: If an operation o is *submitted* to all correct replicas, then each correct replica eventually *nd-delivers* o or $\perp$; if o is deterministic, each correct replica *nd-delivers* o and updates its state via *update*.

The liveness property is concerned with deterministic operations only. There is a chance some non-deterministic operations may be *nd-delivered*; however, due to the total order and correctness guarantees, those non-deterministic operations will not cause any inconsistencies—*which is exactly our goal*.

Note that the notion of BFT-ND also captures conventional BFT-SMR (that supports deterministic operations only).

B. The Strawman Approaches

We present the challenges of transforming a conventional BFT to BFT-ND in an asynchronous model. Our goal in this transformation is to preserve the communication and time complexity of BFT, ensuring that the system performance is not significantly degraded. As mentioned in the introduction, it is unclear how to build BFT-ND for the primary-backup and execute-then-order approaches in the asynchronous model. Thus, we focus on the order-then-execute model.

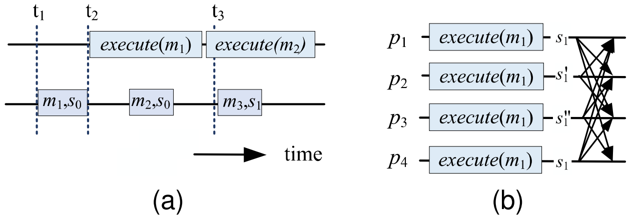


Figure 3. The challenges of building BFT-ND. (a) The dilemma of the 1st attempt in the order-then-execute model. (b) The challenge of using checkpoint for agreement on the state.

A naive approach. A straightforward approach is shown in Figure 3a. Replicas first use a conventional BFT protocol to agree on the order of a block, execute the transactions, and then include the execution results in the order of another block, i.e., the proposed content in the consensus is in the form of (m, s), where m is a block and s is the system state (or the hash of the state). As illustrated in the figure, replicas reach an agreement on the order of a block m_1 (duration $[t_1, t_2]$), execute m_1 in the background, and continue to reach an agreement on the order of other blocks. After m_1 is executed at time t_3, the state s_1 is included in the proposal of another block m_3, i.e., the proposal for m_3 is (m_3, s_1). Here, there is a dilemma on the agreement of s_1 when m_1 consists of non-deterministic operations. In particular, some correct replicas may not maintain s_1 after the execution of m_1. If these correct replicas do not vote for m_3, none of the correct replicas are able to collect more than $2f+1$ matching votes. As $2f + 1$ matching votes are necessary for the agreement on m_3, the protocol suffers from the liveness issue. Alternatively, if correct replicas passively accept s_1, a malicious block proposer can directly manipulate the state of the system, i.e., this design cannot handle the case where m_1 consists of deterministic operations, but the block proposer proposes a *wrong* state.

Note that we can use techniques such as zero-knowledge proof for the replicas to prove the *correctness* of execution results. However, some operations might be expensive to prove [43]. Additionally, such a design modifies the underlying BFT and is thus not modular.

An improved approach via checkpoint protocol. One can optimize the naive approach via the *checkpoint* protocol that is needed for any practical BFT systems. Namely, a checkpoint protocol is designed for garbage collection (i.e., releasing the intermediate consensus parameters stored in memory). A typical workflow is that after a certain number (e.g., b) of blocks are delivered, each replica sends a $\mathsf{checkpoint}()$ message to all replicas. The $\mathsf{checkpoint}()$ message consists of the hash of the highest block and a digital signature for the hash. After collecting $n - f$ matching $\mathsf{checkpoint}()$ messages, the intermediate consensus parameters can be removed from the memory, and the checkpoint becomes *stable*.

If we use the checkpoint protocol for replicas to reach an agreement on the state, we can naturally separate the agreement on the order and the agreement on the state. Namely, the replicas can still reach an agreement on the order in the normal-case operation of the protocol and decide not to agree on the state during the checkpoint protocol.

However, using checkpoint for addressing non-determinism is extremely challenging. Consider an example shown in Figure 3b, let p_1, p_2, p_4 be correct replicas and p_3 be a Byzantine replica. Let $b = 1$, i.e., replicas execute the checkpoint protocol for *every* transaction. Here, we can assume p_1, p_2, and p_4 receive the $\mathsf{checkpoint}()$ messages from each other, so each replica now holds two votes ($\mathsf{checkpoint}()$ messages) for $h_1 = hash(s_1)$ and one vote for $h'_1 = hash(s'_1)$. The faulty replica p_3 may send s_1 to p_1 and p_2 so p_1 and p_2 each has a valid checkpoint ($2f + 1$ matching $\mathsf{checkpoint}()$ messages). However, as p_4 does not receive $2f+1$ matching $\mathsf{checkpoint}()$ messages, it cannot complete the checkpoint protocol.

C. Overview of Our Approach

The idea in Block-ND is de-coupling the agreement on the order of the transactions (the block agreement layer) from the agreement on the state (the state agreement layer), as shown in Figure 2. Namely, replicas first run BFT to order the transactions, execute the transactions, and then reach an agreement on the executed results (states). Obviously, the block agreement layer allows us to reuse existing BFT implementations. For the state agreement layer, our idea is to use Multivalued Byzantine agreement (MBA) [35] that reaches agreement on values from an arbitrary domain; in this way, replicas can decide if they are in consistent states. Recall that MBA guarantees that if *all* correct replicas provide the same input value to MBA (in which case they have the same state), the value will be output by every correct replica. Additionally, if a non-$\perp$ value is decided, at least one correct replica has proposed the value (the non-intrusion property), showing that the corresponding transactions have indeed been executed by at least one correct replica. Accordingly, any correct replica can always obtain the *correct* state and complete the state transfer.

To make the state agreement layer more self-contained, we slightly extend the notion of MBA to a new primitive called double-output MBA (DO-MBA). DO-MBA produces two outputs. The primary output denotes (the hash of) the state replicas reach an agreement on, and the security properties follow those of conventional MBA. The secondary output represents whether a replica needs state transfer. Replicas reach a crusader agreement on the secondary output, i.e., some correct replicas may decide $\perp$ while other correct replicas decide a non-$\perp$ value [39]. If a correct replica decides a non-$\perp$ value for the secondary output, it does not need to perform state transfer—and vice versa.

A practical MBA (and DO-MBA) construction. We provide a novel MBA construction ND-MBA that is faster than existing MBA constructions, as shown in Table I. The main contribution of our construction is using reproposable asynchronous binary agreement (RABA) [16] in a novel manner. Implementing Pisa, the best-known RABA protocol featuring a one-step coin-free fast path [16], our MBA is more efficient than other MBA designs. We further modify a few lines of code to transform our MBA to DO-MBA. To the best of our knowledge, our construction is the first practical MBA protocol ever implemented and evaluated.

Block-ND in a nutshell. Based on DO-MBA, we build Block-ND, an asynchronous and modular system for BFT-ND. Block-ND employs a conventional BFT to order transactions first. After the ordering is finalized, each replica can execute the transactions and provide the hash of its state as input to DO-MBA. We distinguish three different scenarios:

- If the transactions only contain deterministic operations, our approach guarantees that DO-MBA outputs the hash of correct replicas' states (for both the primary output and the secondary output), and the transactions will be *nd-delivered*. (No state transfer is needed.)
- The transactions contain non-deterministic operations, and replicas may still agree on some non-$\perp$ value as the primary output in DO-MBA. In this case, replicas still *nd-deliver* the transactions. Depending on the secondary output in DO-MBA, some correct replicas may start state transfer. Our approach guarantees that correct replicas eventually complete state transfer and converge to the same state.
- The transactions contain non-deterministic operations, and replicas agree on $\perp$ for the primary output in DO-MBA. In this case, all correct replicas roll back to the state before the execution of transactions and then *nd-deliver* $\perp$.

We emphasize that the way that we use rollback and state transfer [44] to handle inconsistent states follows that of CSV [8, pp. 9] (in fact, no solutions can prevent rollback from happening): if a rollback operation is used for execution, a process with a diverging state can obtain the state from other processes via state transfer. Our work does not focus on how to complete state transfer efficiently but studies how to provide an asynchronous and modular treatment to BFT-ND.

To summarize, our paradigm enjoys the following benefits. First, our work is the first practical BFT-ND protocol in the order-then-execute model, as it preserves the complexity of the conventional BFT (for the block agreement layer). As the two layers are executed in parallel, the system performance, as we later show in our evaluation, is only marginally degraded. Second, our work is the first asynchronous treatment of BFT-ND. An interesting fact is that one can even use the state agreement layer (i.e., MBA) to replace the checkpoint protocol, so this layer can also be used for garbage collection!

Most suitable scenario of Block-ND. Block-ND is a perfect fit for blockchain systems in which: 1) non-determinism is unexpected and unwanted; 2) non-deterministic operations occur not that frequently. Our solution is modular and can be used on top of most BFT-based blockchain systems. However, if non-determinism occur frequently, frequent rollback may be triggered, which is undesirable in any systems.

VI. ND-MBA: Practical MBA and DO-MBA

In this section, we provide our DO-MBA construction. We begin with an efficient MBA construction that terminates in only three steps in the optimistic case, as shown in Table I. In contrast, the most efficient MBA protocol known so far is due to Mostéfaoui and Raynal (MR) [35], which terminates

Table I. Comparison of known asynchronous MBA and DO-MBA protocols. *MR reduces MBA to ABA. Here, we consider Quadratic-ABA [36], the most efficient ABA with a fast path known so far. Quadratic-ABA terminates in 4 steps in the best case and the expected number of steps is 10.

protocols	non-intrusion?	msg	best steps	expected steps
CNV MBA [34]	no	$O(n^3)$	14	23
MR [35]*	yes	$O(n^2)$	8	16
ND-MBA	yes	$O(n^2)$	3	12

in eight steps in the optimistic case. We then show how to transform ND-MBA to DO-MBA.

A. Our MBA Construction

```
01 initialization
02   $pv, rv, \rho \leftarrow \bot$          //proposed value and received value
03   $rd \leftarrow [\bot]^*$                 //set of received values
04 upon $mba\text{-}propose(v)$
05   $pv \leftarrow v$
06   broadcast disperse($v$)                  //disperse() step
07 upon receiving disperse($v$) from $p_j$ for the first time
08   $rd[v] \leftarrow rd[v] + 1$
09 upon receiving $n-2f$ disperse($v$) s.t. $v \neq pv$ and echo($v$)
has not been sent                             //optional echo() step
10   broadcast echo($v$)
12 upon receiving echo($v$) from $p_j$ for the first time and
disperse($v$) is not received from $p_j$
13   $rd[v] \leftarrow rd[v] + 1$
14 loop in the background
15   if $\forall x \neq pv, \Sigma_x rd[x] \geq f+1$
16     $r\text{-}propose(0)$ to RABA$_{id}$
17   if $rd[v] \geq n-f$ and forward() has not been sent
18     $\sigma_i \leftarrow tsign(v)$         // forward() step
19     broadcast forward($v, \sigma_i$)
20   let $w$ be the value s.t. $\forall x$ received by $p_i$, $rd[w] \geq rd[x]$
21   if $\Sigma_x rd[x] - rd[w] \geq f+1$
22     $r\text{-}propose(0)$ to RABA$_{id}$
23 upon receiving $n-f$ matching forward($v, \sigma_j$)
24   $\sigma \leftarrow tcombine(\sigma_j \cdots)$
25   if RABA$_{id}$ is not started, $r\text{-}propose(1)$ to RABA$_{id}$
26   else $r\text{-}repropose(1)$ to RABA$_{id}$
27   $rv \leftarrow v$, $\rho \leftarrow \sigma$
28   if distribute() has not been sent        //distribute() step
29     broadcast distribute($rv, \sigma$)
30 upon receiving distribute($v, \sigma$) such that $tverify(v, \sigma)$ and
$n-f$ forward() messages have been received
31   $rv \leftarrow v$, $\rho \leftarrow \sigma$
32   if RABA$_{id}$ is not started, $r\text{-}propose(1)$
33   else $r\text{-}repropose(1)$ to RABA$_{id}$
34 upon $r\text{-}decide(1)$
35   wait until $rv \neq \bot$ and $mba\text{-}decide(rv)$
36 upon $r\text{-}decide(0)$
37   $mba\text{-}decide(\bot)$
```

Figure 4. ND-MBA construction. The code is for p_i.

Overview. We propose a new protocol based on threshold signatures. The core idea is to reduce the MBA problem to RABA, and we use the Pisa protocol by Zhang and Duan [16]. RABA is a variant of asynchronous binary agreement (ABA) protocol that has a coin-free fast path: the protocol can terminate as fast as only one step and does not require coin-tossing.

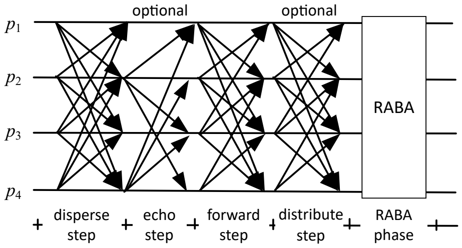


Figure 5. ND-MBA.

ND-MBA only involves several steps of all-to-all communication for replicas to exchange their proposed values. Then a RABA instance is started for replicas to agree on whether a sufficiently large fraction of correct replicas have proposed the same value. In the optimistic case where all correct replicas *mba-propose* the same value, ND-MBA involves two steps of communication and a RABA instance. Hence, ND-MBA terminates in three steps in the fast path and 12 expected steps, much lower than existing ones, as shown in Table I.

Description of the ND-MBA protocol. As shown in Figure 5, our MBA protocol consists of 2-4 communication steps and a RABA instance: disperse(), echo(), forward(), distribute(), and RABA$_{id}$, where RABA$_{id}$ denotes the RABA instance tagged by an identifier id. Briefly speaking, each replica first sends a disperse(v) message to all replicas where v is its proposed value. If a replica receives $n-2f$ disperse(v) where v is different from its proposed value, it sends an echo(v) message to all replicas. These two steps together ensure that every correct replica will receive $n-f$ disperse(v) and echo(v) messages with the same v. Whenever such v exists, the replica sends a forward(v) message to all replicas and can start to propose to RABA$_{id}$ upon receiving a sufficiently large fraction of matching forward() messages. Finally, the distribute() step allows replicas to further exchange their received values from the forward() messages and is used to ensure the special *biased termination* property of RABA.

We show the pseudocode in Figure 4. Every replica p_i maintains four system parameters: pv, rv, ρ, and rd. The pv parameter denotes the *proposed value* of p_i. The value rv stores the *received value*. The ρ parameter stores a *proof* for the value rv, if any. Finally, the rd is a map that tracks the number of received votes for each value.

The protocol proceeds as follows.

▷ *Disperse (lines 04-08).* Upon the $mba\text{-}propose(v)$ event, p_i first sets pv as v, and then broadcasts a disperse(v) message. Meanwhile, every replica uses the rd parameter to track the number of received votes. In particular, upon receiving a disperse(v) message from some replica p_j for the first time (line 07), p_i sets $rd[v]$ as $rd[v]+1$ (line 08).

▷ *Echo (optional, lines 09-13).* If p_i receives $n-2f$ matching disperse(v) messages such that v is different from its proposed value pv, p_i broadcasts an $\mathsf{echo}(v)$ message. Every replica still uses the rd parameter to track the number of received votes in the $\mathsf{echo}()$ messages. If p_i receives an $\mathsf{echo}(v)$ message from p_j for the first time and it has not previously received $\mathsf{disperse}(v)$ from p_j, it sets $rd[v]$ as $rd[v]+1$ (lines 12-13).

$\triangleright$ *Forward (lines 17-20).* Every replica p_i loops in the background and tracks the rd parameter (line 14). If p_i receives $n-f$ matching $\mathsf{disperse}(v)$ and $\mathsf{echo}(v)$ messages, i.e., $rd[v] \geq n-f$ (line 17), p_i creates a partial signature σ_i for value v (line 18) and then broadcasts a $\mathsf{forward}(v, \sigma_i)$ message (line 19). Every correct replica only sends one $\mathsf{forward}()$ message.

$\triangleright$ *Conditions for providing 1 as input to RABA_{id} (lines 23-26, lines 30-33).* There are two conditions.

- Lines 23-26: p_i receives $n-f$ matching $\mathsf{forward}(v, \sigma_j)$ messages. In this case, p_i combines the partial signatures included in the $\mathsf{forward}()$ messages into a signature σ. Then, p_i sets rv as v and ρ as σ (line 27). If p_i has not sent a $\mathsf{distribute}()$ message, it broadcasts a $\mathsf{distribute}(rv, \sigma)$ message (lines 28-29).
- Lines 30-33: p_i receives a valid $\mathsf{distribute}(v, \sigma)$ message such that σ is a valid signature for v. In this case, p_i also sets rv as v and ρ as σ.

$\triangleright$ *Conditions for providing 0 as input to RABA_{id} (lines 15-16, lines 20-22).* There are two conditions.

- Lines 15-16: p_i tracks whether it receives $f+1$ inconsistent $\mathsf{disperse}(v)$ and $\mathsf{echo}(v)$ for any value v different from pv. Once the condition "$\forall x \neq pv$, $\Sigma_x rd[x] \geq f+1$" is satisfied, at least one correct replica must have proposed a value different from pv. In this case, p_i *r-proposes* 0 to RABA_{id}.
- Lines 20-22: p_i tracks value w for which it receives the highest number of votes, i.e., $rd[w] \geq rd[x]$. Then, if the number of votes for all other values is at least $f+1$ higher than $rd[w]$ (i.e., $\Sigma_x rd[x] - rd[w] \geq f+1$), p_i *r-proposes* 0 to RABA_{id}.

$\triangleright$ *Output conditions (lines 34-37).* Every replica waits for RABA_{id} to terminate. There are two cases. If p_i *r-decides* 1, it waits for rv to become non-$\perp$. After that, it *mba-decides* rv. Otherwise, if p_i *r-decides* 0, it *mba-decides* $\perp$.

Complexity analysis. ND-MBA only involves all-to-all communication and the message complexity of known RABA protocols (e.g., Pisa [16]) is $O(n^2)$. Hence, the message complexity of ND-MBA is $O(n^2)$. The time complexity is $O(1)$ as every phase completes in constant time. We now analyze the communication complexity. Consider that the input of each replica is L. Replicas exchange their proposed values in the $\mathsf{disperse}()$ and $\mathsf{echo}()$ steps, so the communication complexity is $O(Ln^2)$. In the $\mathsf{forward}()$ and $\mathsf{disperse}()$ steps, each replica sends one value and a signature to all replicas, so these two steps have $O(Ln^2 + \kappa n^2)$ communication, where κ is the length of the security parameter (e.g., the length of the signature). In the RABA phase, the communication is $O(\kappa n^2)$, considering that the common coin is instantiated by threshold PRF [45]. Therefore, ND-MBA has $O(Ln^2 + \kappa n^2)$ communication.

B. Formalizing DO-MBA

We now formally define DO-MBA that extends MBA. As mentioned in Sec. V-C, MBA with the non-intrusion property already ensures that replicas will eventually agree on the same state. We slightly extend the notion to DO-MBA mainly because DO-MBA is a self-contained notion for the agreement on the state. Namely, the needs for state transfer is directly exposed to users as part of the output.

In DO-MBA, every correct replica *mba-proposes* one value $v \in \{0,1\}^L$ and *mba-decides* two values (v_1, v_2), where L is a finite integer. Here v_1 and v_2 are called the primary output and the secondary output, respectively. Both the primary output and the secondary output can be $\perp$ (a distinguished symbol). We require the conventional agreement property for the primary output and weak agreement (as in the crusader agreement) for the secondary output. In particular, DO-MBA satisfies the following properties:

- **Validity.** If all correct replicas *mba-propose* v_1, all correct replicas eventually *mba-decide* (v_1, v_2) for any v_2.
- **Primary agreement.** If a correct replica *mba-decides* (v_1, v_2) and a correct replica *mba-decides* (v_1', v_2') such that $v_1 \neq \perp$ and $v_1' \neq \perp$, then $v_1 = v_1'$.
- **Weak secondary agreement.** If a correct replica *mba-decides* (v_1, v_2) and a correct replica *mba-decides* (v_1', v_2'), then $v_2 = v_2'$ or one of v_2 and v_2' is $\perp$.
- **Termination.** If all correct replicas *mba-propose*, every correct replica eventually *mba-decides* some value.
- **Integrity.** Every correct replica *mba-decides* once.
- **Non-intrusion**. If a correct replica *mba-decides* (v_1, v_2), at least one correct replica *mba-proposes* v_1.

The DO-MBA primitive has features for the agreement on the state, considering the input of each replica is the hash of its state. First, if all correct replicas *mba-propose* the same value v_1, the validity property of DO-MBA guarantees that all correct replicas will *mba-decide* (v_1, v_2) (in our construction, $v_1 = v_2$). Hence, if all correct replicas execute non-deterministic operations in the same order, they will *mba-propose* the same value v_1, *mba-decide* (v_1, v_2), and do not need state transfer. Second, the secondary output of DO-MBA captures the feature for state transfer. In particular, any replica that *mba-decides* $(v_1, \perp)$ will start state transfer. The non-intrusion property of DO-MBA guarantees that at least one correct replica *mba-proposes* v_1 and the hash of its state is v_1. Hence, all correct replicas will eventually complete the state transfer.

We intentionally choose to not define any validity property for the secondary output. Jumping ahead, our construction achieves a validity property for the secondary output as follows: If all correct replicas *mba-propose* v_1, all correct replicas eventually *mba-decide* (v_1, v_2) where $v_2 = v_1$. However, our abstraction does not need such a validity property by design. We believe such an abstraction provides some flexibility for deployment of our protocols.

C. Our DO-MBA Construction

We transform ND-MBA to a DO-MBA protocol by replacing lines 34-37 with ones shown in Figure 6. In particular,

```
replace lines 34-37 in Figure 4 using the following lines
34 upon r-decide$(1)$
35   wait until $rv \neq \bot$
36   if $rv = pv$
37     mba-decide$(rv, rv)$
38   else
39     mba-decide$(rv, \bot)$
40 upon r-decide$(0)$
41   mba-decide$(\bot, \bot)$
```

Figure 6. Transforming ND-MBA to DO-MBA.

each replica p_i waits for RABA_{id} to terminate. There are two cases. First, p_i *r-decides* 1. Here, replicas have already reached an agreement on some value for the primary output. Replica p_i then waits for its rv to become non-$\bot$ (line 35). This rv value is updated in the $\mathsf{forward}()$ and $\mathsf{distribute}()$ steps. After that, p_i verifies whether rv is the same as its proposed value pv. If so, p_i *mba-decide*(rv, rv) (lines 36-37). Otherwise, p_i *mba-decide*$(rv, \bot)$ (lines 38-39). Second, if p_i *r-decides* 0, p_i *mba-decide*$(\bot, \bot)$ (lines 40-41).

Complexity analysis. Our transformation from the MBA protocol to the DO-MBA protocol only involves additional local computation. Therefore, our DO-MBA protocol preserves the complexity of ND-MBA, achieving $O(1)$ time, $O(n^2)$ messages, and $O(Ln^2 + \kappa n^2)$ communication. When we use our DO-MBA protocol in Block-ND, the input of each replica is always a hash, so the communication complexity is $O(\kappa n^2)$. We show the proof of our protocol in our full paper [46].

VII. Block-ND

This section describes Block-ND consisting of a block agreement layer ordering transactions and a state agreement layer reaching an agreement on the state. We present in Figure 7 the workflow of Block-ND. For the block agreement layer, we use the *a-deliver*(sn, m) event, i.e., replicas *a-deliver* m and explicitly assign a sequence number sn to m. For DO-MBA, we use the *mba-propose*() and *mba-decide*() events.

Every replica p_i maintains a state s. We use s_{sn-1} to denote the state before querying the *execute*(s_{sn-1}, m) function and s_{sn} to denote the state after the execution of m.

The protocol works as follows. After the *a-deliver*(sn, m) event (line 03), each replica p_i executes the transactions in block m by querying the function *execute*(s_{sn-1}, m), and obtains the state s_{sn} (line 05). Then at line 06, p_i starts a DO-MBA instance MBA_{sn} and provides $hash(s_{sn})$ as the input.

There are three cases after MBA_{sn} outputs (v, h) (line 07).

- Lines 08-09: $v \neq \bot$ and $h \neq \bot$. In this case, replicas reach an agreement on $hash(s_{sn})$ and the state of p_i matches one that replicas reach an agreement on. Replica p_i then *nd-delivers* m.
- Lines 10-12: $v \neq \bot$ and $h = \bot$. In this case, replicas reach an agreement on the state, such that the hash of the state is v. Additionally, p_i maintains an inconsistent state with other correct replicas. In this case, p_i performs state transfer with all the replicas until it updates it state, the hash of which is h. Then p_i *nd-delivers* m.

```
01 initialization
02   $s, msg$      //$s$ denotes state and $msg$ is a delivered block
03 upon a-deliver$(sn, m)$                          //block agreement
04   $msg_{sn} \leftarrow m$
05   $s_{sn} \leftarrow$ execute$(s_{sn-1}, m)$                         //execution
06   mba-propose$(hash(s_{sn}))$ for $\mathsf{MBA}_{sn}$   //state agreement
07 upon mba-decide$(v, h)$ for $\mathsf{MBA}_{sn}$
08   if $v \neq \bot$ and $h \neq \bot$
09     nd-deliver$(msg_{sn})$ such that $hash(s_{sn}) = v$
10   if $v \neq \bot$ and $h = \bot$                        //state transfer
11     perform state transfer until $hash(s_{sn}) = v$
12     nd-deliver$(msg_{sn})$ such that $hash(s_{sn}) = v$
13   if $v = \bot$        //$m$ contains non-deterministic operations
14     $s_{sn} \leftarrow s_{sn-1}$                                   //rollback
15     nd-deliver$(\bot)$
```

Figure 7. The workflow of Block-ND. The code for p_i.

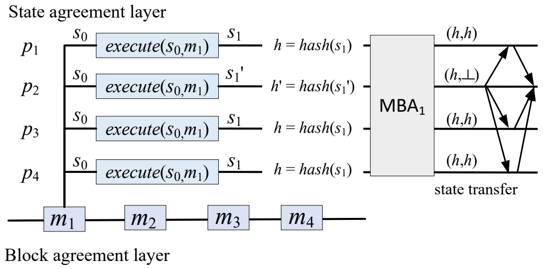


Figure 8. A running example of Block-ND.

- Lines 13-14: $v = \bot$. In this case, replicas fail to reach an agreement on the same state. Alternatively, we can also say that replicas reach an agreement on the fact that the block m consists of at least one transaction with non-deterministic operations. Replica p_i then rolls back to the state of the prior block, i.e., by setting s_{sn} as s_{sn-1}. It then *nd-delivers* a special symbol $\bot$.

A running example. We illustrate a running example of Block-ND in Figure 8. All correct replicas initially maintain state s_0. After the order of block m_1 is finalized (i.e., an agreement on the order is reached), replicas use the hash of the state as input to MBA_1, a DO-MBA instance. There are three possible scenarios:

(1) Block m_1 does not include any transactions with non-deterministic operations. Accordingly, all correct replicas maintain the same state s_1 after *execute*(s_0, m_1). According to the validity property of DO-MBA, all correct replicas will *mba-decide* (h, h) where $h = hash(s_1)$. No replicas need to perform state transfer. All correct replicas then *nd-deliver* m_1 and their state is s_1.

(2) Block m_1 contains transactions with non-deterministic operations, but correct replicas still reach an agreement on some state. An example is shown in Figure 8 with four replicas. Replicas p_1, p_3, and p_4 obtain the same execution result, and their state is s_1. Replicas p_1, p_3, and p_4 provide h as input to DO-MBA but replica p_2 obtains s_1' and provides h' as input. After DO-MBA terminates, p_1, p_3, and p_4 *mba-decide* (h, h) and p_2 *mba-decides* $(h, \bot)$. Then

p_2 performs state transfer with other replicas. According to the non-intrusion property of DO-MBA, the protocol guarantees that at least one correct replica maintains the state s_1 such that $h = hash(s_1)$ so p_3 can successfully complete the state transfer. After that, correct replicas then *nd-deliver* m_1 and their state is s_1.

(3) The block m_1 contains transactions with non-deterministic operations and correct replicas reach an agreement on $(\perp, \perp)$. All correct replicas then roll back to the state s_0 and *nd-deliver* $\perp$.

We prove the correctness of Block-ND in our full paper [46].

Why order-then-execute? Applicability to other models. In Block-ND, as the agreement on the order and the agreement on the state are de-coupled, the agreement on the state can be triggered in the background. We show in our experiments that by doing so, this paradigm creates little overhead to the system performance.

The DO-MBA primitive itself can be used for other models as well. For instance, in the execute-then-order model, replicas can execute the transactions and use DO-MBA to agree on the execution result. If DO-MBA outputs a non-$\perp$ value for the primary output, replicas then reach a consensus on the order of the corresponding transactions.

Handling transactions with non-deterministic operations. To build a fully-fledged BFT-ND protocol, we still need to consider how to handle scenario (3) mentioned above: correct replicas *nd-deliver* $\perp$ for block m (where m is *a-delivered*). As mentioned above, replicas have already agreed on the fact that m contains at least one transaction with non-deterministic operations. According to the protocol, m should directly be discarded by the replicas.

However, discarding the entire block creates a subtle liveness issue. In particular, m consists of multiple transactions. Consider that only one transaction o in m consists of non-deterministic operations, while other transactions only contain deterministic operations. If correct replicas directly *nd-deliver* $\perp$ and discard m, all deterministic transactions in m will be discarded, violating the liveness property of BFT-ND.

To address this issue, we further make the following change to Block-ND. After each correct replica *nd-delivers* $\perp$ for any block m, replicas do not immediately discard all the transactions in m. Instead, replicas execute the transactions in m sequentially (in a deterministic order) and start an MBA instance for each transaction. After the MBA instance terminates, replicas handle the transaction in exactly the same way as described above. In this way, transactions with deterministic operations can then be *nd-delivered*, and the liveness property is satisfied. Such an approach, while being expensive, seems unavoidable. We consider the optimization a future work.

VIII. Implementation and Evaluation

We implement a prototype of Block-ND in Golang. Our implementation of the protocols involves more than 11,000 LOC. For the block agreement layer, we implement both PACE [16] (an asynchronous BFT protocol) and PBFT [17] (a partially synchronous BFT protocol). We use gRPC as the communication library. We use HMAC to realize the authenticated channel, SHA256 as the hash function, and ECDSA as the digital signature scheme. For the threshold signature scheme (used in our DO-MBA protocol), we use a set of ECDSA signatures instead, following that of a large number of prior systems [18, 19]. For the *execute*() function, we use the open-source EVM implementation from the Hyperledger Burrow project[1]. We evaluate the performance on Amazon EC2 using up to 91 virtual machines (VMs). We use m5.xlarge instances. The m5.xlarge instance has four virtual CPUs and 16GB memory. We deploy our protocols in the WAN setting, where replicas are evenly distributed across the following regions: us-west-2 (Oregon, US), us-east-2 (Ohio, US), ap-southeast-1 (Singapore), and eu-west-1 (Ireland).

We conduct the experiments under different network sizes and batch sizes. We use f to denote the network size and we use $n = 3f+1$ replicas in each experiment. We use b to denote the batch size, where each replica proposes b transactions at a time. The default transaction size is 250 bytes. For each experiment, we repeat the experiment 5 times and report the average performance result.

In terms of performance comparison, we also use FISCO-BCOS [47][2] and Hyperledger Fabric[3] (Fabric for short). We also implement a simple CSV framework to understand its performance. Our evaluation aims to answer the following questions:

- How efficient is our DO-MBA protocol?
  - How does our DO-MBA protocol perform under different input conditions (i.e., correct replicas provide the same input and inconsistent inputs)?
  - What is the latency breakdown for our DO-MBA protocol?
- How does Block-ND perform compared to conventional BFT protocols? What is the performance overhead introduced by running an additional DO-MBA protocol?
- What is the performance of Block-ND under failures?

Performance of our ND-MBA protocol. In our constructions, the input to the DO-MBA is a fixed-length hash. Therefore, we assess the latency of our DO-MBA protocol for different f. To better analyze the performance overhead, we assess four different modes of DO-MBA protocol for each f.

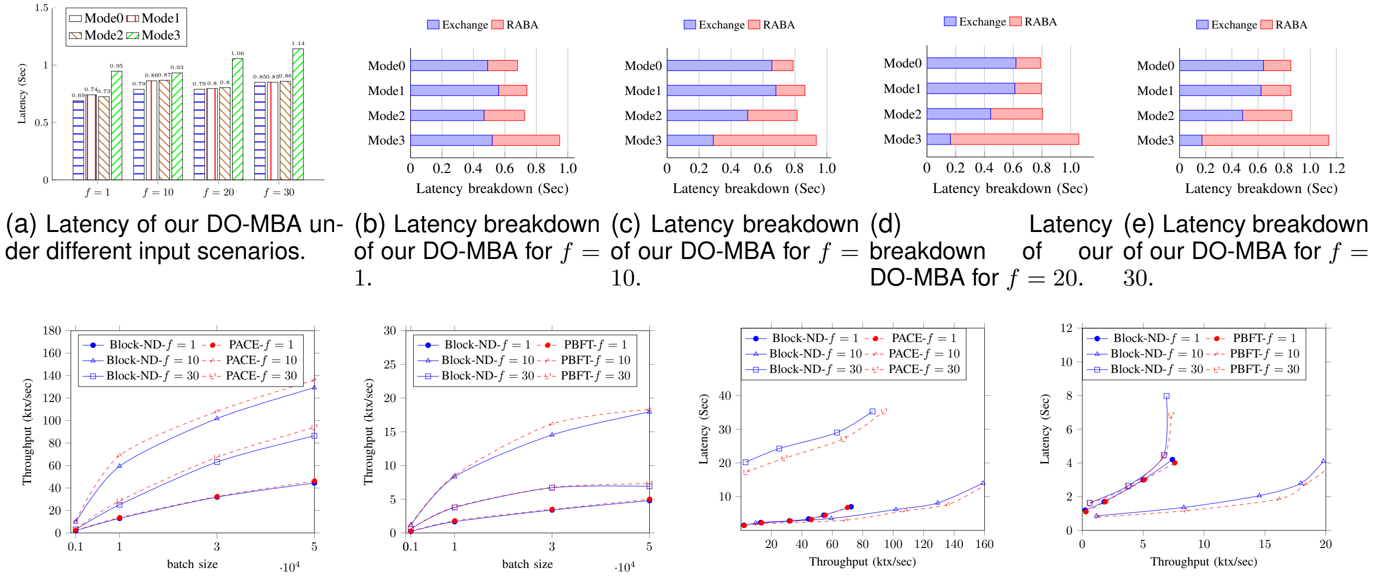
- Mode 0: All correct replicas provide the same input.
- Mode 1: $2f$ correct replicas provide the same input, and one correct replica provides an inconsistent input.
- Mode 2: $f+1$ correct replicas provide the same input while f correct replicas provide some inconsistent inputs.
- Mode 3: Every correct replica generates a local random value and provides the value as the input.

We show the latency of our DO-MBA protocol for $f = 1, 10, 20, 30$ under the four modes in Figure 9a. All the experiments are completed within 1.14 seconds, where the experiment with the highest latency is for $f = 30$ (91 replicas) and mode 3. Among the four modes, the latency of mode 3

[1] Burrow: https://github.com/hyperledger-archives/burrow

[2] FISCO-BCOS: https://github.com/FISCO-BCOS/FISCO-BCOS/blob/master/docs/README_EN.md

[3] Fabric: https://github.com/hyperledger/fabric



(a) Latency of our DO-MBA under different input scenarios.

(b) Latency breakdown of our DO-MBA for $f = 1$.

(c) Latency breakdown of our DO-MBA for $f = 10$.

(d) Latency breakdown of our DO-MBA for $f = 20$.

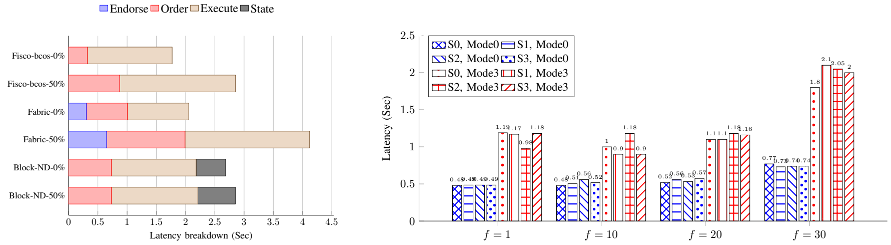
(e) Latency breakdown of our DO-MBA for $f = 30$.

(f) Throughput of Block-ND (PACE) and PACE.

(g) Throughput of Block-ND (PBFT) and PBFT.

(h) Latency v.s. Throughput for Block-ND (PACE) and PACE.

(i) Latency v.s. Throughput for Block-ND (PBFT) and PBFT.



f	PACE	Block-ND \Degradation (PACE)	CSV (PACE)
1	30.00	29.51 \ 1.65%	3.01
10	87.30	79.4 \ 9.94%	3.39
f	**PBFT**	**Block-ND \Degradation (PBFT)**	**CSV (PBFT)**
1	11.77	10.88 \ 8.23%	2.85
10	13.91	12.74 \ 7.24%	3.11

(j) Latency breakdown of FISCO-BCOS, Hyperledger Fabric and Block-ND for each transaction.

(k) Latency of our DO-MBA in Block-ND (PACE) under different failure scenarios.

(l) Throughput (ktx/sec) Block-ND and CSV, and the performance degradation of Block-ND.

Figure 9. Evaluation results of our DO-MBA protocol, Block-ND and other benchmarks.

is consistently higher than the other three modes. To further assess the result, we also present the latency breakdown in Figure 9b-9e. We use "exchange" to denote the disperse(), echo(), forward(), and distribute() steps, and "RABA" to denote the RABA phase. As shown in the figures, the RABA phase has a higher latency in mode 3 than in the other three modes. For instance, when $f = 30$, the RABA phase occupies 87.3% of the total runtime in mode 3, in contrast to 32.5% in mode 1. The results are expected, as in mode 3, it is more likely that replicas will provide 0 as input to RABA in the early stage of the exchange phase. In such a case, RABA will terminate in more rounds.

Performance of Block-ND. We assess Block-ND using PACE and PBFT as the block agreement layer, denoted as Block-ND (PACE) and Block-ND (PBFT), respectively. We compare the performance under two scenarios: running a PACE (resp. PBFT) instance and running Block-ND (PACE) (resp. Block-ND (PBFT)). In this way, we can evaluate the overhead created by our DO-MBA protocol. We assess three different scenarios.

- *Non-deterministic benchmark*. We construct a benchmark with both deterministic and non-deterministic contracts and compare the performance of FISCO-BCOS, Hyperledger Fabric, and Block-ND. For this scenario, we aim to understand the performance of different platforms in terms of handling non-deterministic transactions.
- *No execution benchmark*. We neglect the cost of execution. For this scenario, we aim to understand the performance of Block-ND itself.
- *Smart contract benchmark*. We assess the throughput using only deterministic contracts. For this scenario, we also assess the performance of CSV to understand the performance under the execute-then-order and order-then-execute model.

Non-deterministic benchmark. We construct two benchmarks: 0% non-deterministic contracts; 50% non-deterministic transactions. FISCO-BCOS uses an implementation of EVM called evmone[4], Fabric uses Chaincode, and our implementa-

[4] evmone: https://github.com/ethereum/evmone

tion uses EVM. As identifying non-deterministic transaction is orthogonal to the problem our study, we *construct* the non-deterministic transactions and *make* execution results by correct replicas inconsistent. The benchmark consists of 1,000 transactions, and we summarize the average latency breakdown of each transaction for three systems in Figure 9j. Our results show that Block-ND outperforms both FISCO-BCOS and Fabric under the 50% non-deterministic benchmark.

No execution benchmark. We demonstrate the throughput for $f = 1, 10, 30$, varying the batch size for BFT in the block agreement layer. We report batch size vs. throughput in Figure 9f-9g and throughput vs. latency in Figure 9h-9i. In our experiments, the performance of Block-ND degrades marginally compared to that of running a single BFT instance. In particular, the throughput of Block-ND (PBFT) degrades 0.89%-10.02% compared to that of PBFT and the throughput of Block-ND (PACE) degrades 1.47%-11.79% compared to that of PACE. The difference between Block-ND and a single BFT instance is more visible when f is large. As for the throughput vs. latency, given the same throughput, the latency of PACE (resp. PBFT) is consistently and slightly lower than that of Block-ND (PACE) (resp. Block-ND (PBFT)).

Smart contract benchmark. We use EVM as the execution layer and PACE/PBFT as the consensus to implement CSV. For example, in the EVM+PBFT combination of CSV, after the leader collects $f+1$ matching execution results, it proposes the transaction. We report the throughput of both the block agreement layer and the state agreement layer for $f = 1$ and 10 using only deterministic transactions. We fix the batch size to 30,000. We study three different scenarios and assess the throughput: PACE/PBFT with EVM, Block-ND with EVM, and CSV. We summarize our results in Figure 9l.

Our evaluation results show that the throughput of the state agreement layer is 3.5 ktx/sec (on average for almost all experiments), and the throughput of CSV is from 2.85-3.39 ktx/sec. Compared with the throughput of the block agreement layer (e.g., 33.0 ktx/sec for PACE), the throughput of the state agreement layer and CSV are significantly lower.

In Block-ND, the block agreement layer does not need to wait for the state agreement layer or the execution of the smart contract to complete before starting a new epoch. Therefore, the performance of the block agreement layer with EVM execution degrades only marginally. As shown in Figure 9l, for $f = 1$ and $f = 10$, the throughput degradation of Block-ND (PACE) are 1.6% and 9.94%; the throughput degradation of Block-ND (PBFT) is 8.23% and 7.24%. In all cases with EVM executions, the performance bottleneck of Block-ND is due to the executions in the state agreement layer.

Performance under failures. We assess the performance of Block-ND (PACE) for $f = 1, 10, 20, 30$ under four different scenarios, following the practice in prior work [16, 36].

- S0: All replicas are correct.
- S1: Let f replicas crash by not processing any message.
- S2: Let f faulty replicas keep voting 0 in RABA in both PACE and DO-MBA.
- S3: Let f replicas fail by always voting for the flipped value in RABA in both PACE and DO-MBA.

We report the latency of our DO-MBA protocol in Figure 9k. For mode 0, the latency of DO-MBA in the four scenarios is almost identical. This is because correct replicas all vote for 1 in RABA, so faulty replicas cannot render RABA to terminate in a larger number of rounds. For mode 3, the latency of the DO-MBA protocol varies slightly for different scenarios. For larger f, the latency under failure scenarios is slightly higher compared to the failure-free scenario. We conclude that the performance of our DO-MBA protocol is dominated by the inputs of the replicas.

IX. Conclusion

We revisit the notion of Byzantine fault-tolerant state machine replication with non-determinism (BFT-ND) and build an efficient, modular, and asynchronous system called Block-ND. At the core of Block-ND is a novel idea of separating agreement on transaction ordering from agreement on replica state. As a key building block for Block-ND, we formalize a new distributed computing primitive—DO-MBA and provide an efficient construction. Our evaluation results show that Block-ND incurs marginal overhead to the conventional BFT systems dealing with deterministic operations only.

References

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *EuroSys*, 2018.

[2] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, 2017.

[3] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[4] Marko Vukolić. Rethinking permissioned blockchains. In *BCC*, pages 3–7, 2017.

[5] Solidity documentation. https://docs.soliditylang.org/en/latest/, 2022.

[6] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *ESEC/FSE*, pages 1110–1114, 2019.

[7] Enis Ceyhun Alp, Eleftherios Kokoris-Kogias, Georgia Fragkouli, and Bryan Ford. Rethinking general-purpose decentralized computing. In *HotOS*, pages 105–112, 2019.

[8] Christian Cachin, Simon Schubert, and Marko Vukolić. Non-determinism in byzantine fault-tolerant replication. *OPODIS*, 2016.

[9] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *SOSP*, volume 35, pages 15–28. ACM, 2001.

[10] Rüdiger Kapitza, Matthias Schunter, Christian Cachin, Klaus Stengel, and Tobias Distler. Storyboard: Optimistic deterministic multithreading. In *HotDep*, 2010.

[11] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *SOSP*, pages 45–58, 2005.

[12] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *OSDI*, pages 237–250, 2012.

[13] Sisi Duan and Haibin Zhang. Practical state machine replication with confidentiality. In *SRDS*, pages 187–196. IEEE, 2016.

[14] Hyperledger Fabric documentation. https://hyperledger-fabric.readthedocs.io/_/downloads/vi/latest/pdf/, 2023.

[15] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, 1988.

[16] Haibin Zhang and Sisi Duan. Pace: Fully parallelizable bft from reproposable byzantine agreement. In *CCS*, 2022.

[17] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *TOCS*, 20(4):398–461, 2002.

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

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

[20] Sisi Duan and Haibin Zhang. Foundations of dynamic bft. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1317–1334, 2022.

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

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

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

[24] Sisi Duan, Xin Wang, and Haibin Zhang. Practical signature-free asynchronous common subset in constant time. *ACM CCS*, 2023.

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

[26] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[27] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS*, pages 254–269, 2016.

[28] Chaincode scanner tool. https://chainsecurity.com/, 2022.

[29] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *SOSP*, 37(5):253–267, 2003.

[30] Russell Turpin and Brian A. Coan. Extending binary byzantine agreement to multivalued byzantine agreement. *Inf. Process. Lett.*, 18(2):73–76, 1984.

[31] Valerie King and Jared Saia. Breaking the $o(n^2)$ bit barrier: scalable Byzantine agreement with an adaptive adversary. *JACM*, 58(4):18, 2011.

[32] G. Liang and N. Vaidya. Error-free multi-valued consensus with byzantine failures. In *PODC*, 2011.

[33] Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued byzantine agreement. In *PODC*, pages 163–168, 2006.

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

[35] Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous byzantine systems: from multivalued to binary consensus with t< n/3, o(n^2) messages, and constant time. *Acta Informatica*, 54(5):501–520, 2017.

[36] Haibin Zhang, Sisi Duan, Boxin Zhao, and Liehuang Zhu. Waterbear: Practical asynchronous bft matching security guarantees of partially synchronous bft. 2023.

[37] Haibin Zhang, Sisi Duan, Chao Liu, Boxin Zhao, Xuanji Meng, Shengli Liu, Yong Yu, Fangguo Zhang, and Liehuang Zhu. Practical asynchronous distributed key generation: Improved efficiency, weaker assumption, and standard model. IEEE DSN, 2023.

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

[39] Danny Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

[40] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, 2003.

[41] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, 2000.

[42] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, volume 93, pages 42–51. Citeseer, 1993.

[43] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *CCS*, pages 955–966. ACM, 2013.

[44] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. From byzantine replication to blockchain: Consensus is only the beginning. In *DSN*, pages 424–436. IEEE, 2020.

[45] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[46] Yue Huang, Huizhong Li, Yi Sun, and Sisi Duan. Byzantine fault tolerance with non-determinism, revisited. Cryptology ePrint Archive, Paper 2024/134, 2024. https://eprint.iacr.org/2024/134.

[47] Huizhong Li, Yujie Chen, Xiang Shi, Xingqiang Bai, Nan Mo, Wenlin Li, Rui Guo, Zhang Wang, and Yi Sun. Fisco-bcos: An enterprise-grade permissioned blockchain system with high-performance. In *SC*, pages 1–17, 2023.

Appendix

We prove the correctness of our DO-MBA. As our DO-MBA already implies an MBA protocol, correctness of our MBA follows.

Theorem 1 (Non-intrusion)**.** *If a correct replica mba-decides (v_1, v_2), at least one correct replica mba-proposes v_1.*

Proof: If a correct replica *mba-decides* (v_1, v_2), at least one correct replica sets rv as v_1 and ρ as σ, where σ is a valid signature for v_1. Hence, at least $n-2f$ correct replicas have sent $\mathsf{forward}(v_1, -)$. For any correct replica that sends a $\mathsf{forward}(v_1, -)$ message, it has received $n-f$ $\mathsf{echo}(v_1)$ and $\mathsf{disperse}(v_1)$ messages. According to the protocol, if one correct replica receives an $\mathsf{echo}(v_1)$ message, it has received $f+1$ $\mathsf{disperse}(v_1)$ messages. Thus, at least one correct replica sends a $\mathsf{disperse}(v_1)$ message and *mba-proposes* v_1. ■

Lemma 2. *If a correct replica r-proposes* 1 *or r-reproposes* 1*, it sets rv as v and ρ as σ where σ is a valid signature for v. If another correct replica r-proposes* 1 *or r-reproposes* 1*, it sets rv as v and ρ as σ.*

Proof: If a correct replica *r-proposes* 1 or *r-reproposes* 1, it has received $n-f$ matching $\mathsf{forward}(v, -)$ messages or a $\mathsf{distribute}(v, -)$ message. Replica p_i sets ρ as σ where σ is a valid signature for v. We assume that another correct replica p_j sets rv as v' and prove the correctness by contradiction. In particular, if p_j *r-proposes* 1 or *r-reproposes* 1, it has received $n-f$ matching $\mathsf{forward}(v', -)$ or a valid $\mathsf{distribute}(v', \sigma')$ message where σ' is a valid signature for v'. Therefore, at least one correct replica must have sent both $\mathsf{forward}(v, -)$ and $\mathsf{forward}(v', -)$, contradicting the fact that every correct replica only sends a $\mathsf{forward}()$ message once. ■

Lemma 3. *If correct replicas r-decide* 1*, at least one correct replica sends a* $\mathsf{distribute}(v, \sigma)$ *message, where σ is a valid signature for v. If any replica receives* $\mathsf{distribute}(v', \sigma')$*, $v = v'$.*

Proof: If correct replicas *r-decide* 1, at least one correct replica *r-proposes* 1 or *r-reproposes* 1, as otherwise the validity property of RABA is violated. According to Lemma 2, at least one correct replica sets rv as v and ρ as σ such that σ is a valid signature for v. If another replica sends $\mathsf{distribute}(v', \sigma')$ such that σ' is a valid signature for v', at least one correct replica must have sent both $\mathsf{forward}(v, -)$ and $\mathsf{forward}(v', -)$. As every correct replica only sends a $\mathsf{forward}()$ message once, $v = v'$. ■

Theorem 4 (Validity)**.** *If all correct replicas mba-propose* v_1*, all correct replicas eventually mba-decide* (v_1, v_2) *for any* v_2*.*

Proof: If all correct replicas *mba-propose* v_1, no correct replicas will receive more than $f + 1$ $\mathsf{disperse}(v_1')$ messages s.t. $v_1 \neq v_1'$. Therefore, no correct replica will *r-propose* 0. Every correct replica eventually receives $n-f$ $\mathsf{disperse}(v_1)$ and then broadcasts $\mathsf{forward}(v_1, -)$. Similarly, no correct replica will receive $\mathsf{forward}(v_1', -)$ as a valid σ requires $n - f$ partial signatures for v_1'. Furthermore, no correct replica will send an $\mathsf{echo}(v_1')$ message. The condition $\Sigma_x rd[x] - rd[v] \geq f + 1$ will not be triggered so no correct replica will *r-propose* 0. Similarly, the condition $\forall x \neq pv$, $\Sigma_x rd[x] \geq f+1$ will not be triggered as no replica is able to receive $f + 1$ $\mathsf{disperse}(v_1')$.

As every correct replica will send a $\mathsf{forward}(v_1, \sigma_i)$ message, every correct replica will eventually receive $n - f$ $\mathsf{forward}(v_1, \sigma_j)$ and *r-proposes* 1. According to the biased validity property of RABA, every correct replica eventually *r-decides* 1. According to the protocol, any correct replica that *mba-proposes* v sets pv as v_1. Additionally, every replica sets rv as v_1 and pv as v_1, it will then *mba-decide* (v_1, v_1). The theorem thus holds. ■

Theorem 5 (Primary agreement)**.** *If a correct replica mba-decides* (v_1, v_2) *and a correct replica mba-decides* (v_1', v_2') *such that* $v_1 \neq \bot$ *and* $v_1' \neq \bot$*, then* $v_1 = v_1'$*.*

Proof: We assume $v_1 \neq v_1'$ and prove the theorem by contradiction. According to the protocol, any correct replica that *mba-decides* must have *r-decided* 1. If a correct replica p_i *mba-decides* (v_1, v_2), there are two cases: 1) p_i receives $n - f$ $\mathsf{forward}(v_1, -)$ messages, it sets rv as v_1 and ρ as σ where σ is a valid signature for v_1; 2) p_i receives a $\mathsf{distribute}(v_1, \sigma)$ message from another replica such that σ is a valid signature for v_1. If another correct replica p_j *mba-decides* (v_1', v_2'), p_j also receives a valid signature for v_1' after receiving $n-f$ $\mathsf{forward}(v_1', -)$ messages or a $\mathsf{distribute}(v_1', -)$ message. Hence, at least one correct replica must have sent both $\mathsf{forward}(v_1, -)$ and $\mathsf{forward}(v_1', -)$, contradicting the fact that every correct replica only sends a $\mathsf{forward}()$ message once. ■

Theorem 6 (Weak secondary agreement)**.** *If a correct replica mba-decides* (v_1, v_2) *and a correct replica mba-decides* (v_1', v_2')*, then* $v_2 = v_2'$ *or one of* v_2 *and* v_2' *is* $\bot$*.*

Proof. If a correct replica *mba-decides* (v_1, v_2), there are three cases: 1) $v_1 = \bot$. In this case $v_2 = \bot$ according to the protocol; 2) $v_1 \neq \bot$ and $v_2 \neq \bot$. According to the protocol, $v_1 = v_2$; 3) $v_1 \neq \bot$ and $v_2 = \bot$. Similarly, if another correct replica *mba-decides* (v_1', v_2'), there are three cases: 1) $v_1' = \bot$ and $v_2' = \bot$; 2) $v_1' \neq \bot$, $v_2' \neq \bot$, and $v_1' = v_2'$; 3) $v_1' \neq \bot$ and $v_2' = \bot$. We show that for every combination of v_1, v_1', v_2', and v_2', either $v_2 = v_2'$ or at least v_2 or v_2' is $\bot$.

- $v_1 = \bot$. For all the three cases for v_1' and v_2', the theorem holds as $v_1 = v_2 = \bot$.
- $v_1 \neq \bot$ and $v_2 \neq \bot$; $v_1' = \bot$. The theorem holds as $v_1' = \bot$.
- $v_1 \neq \bot$ and $v_2 \neq \bot$; $v_1' \neq \bot$ and $v_2' \neq \bot$. According to the protocol, $v_1 = v_2$ and $v_1' = v_2'$. According to the primary agreement property, we know that $v_1 = v_1'$. Therefore, $v_2 = v_2'$.
- $v_1 \neq \bot$ and $v_2 \neq \bot$; $v_1' \neq \bot$ and $v_2' = \bot$. The theorem holds as $v_2' = \bot$.
- $v_1 \neq \bot$ and $v_2 = \bot$. For all three cases for v_1' and v_2', the theorem holds as $v_2 = \bot$. ■

Lemma 7. *If a correct replica r-proposes* 1 *or r-reproposes* 1*, any correct replica either r-proposes* 1 *or will later r-repropose* 1*.*

Proof: If a correct replica *r-proposes* 1, it has received $n - f$ matching $\mathsf{forward}(v, -)$ messages and set ρ as σ where σ is a valid signature for v. The replica will broadcast a $\mathsf{distribute}(v, \sigma)$ message. According to the protocol, any correct replica that receives a $\mathsf{distribute}(v, \sigma)$ message either has *r-proposed* 1 or will *r-repropose* 1. ■

Theorem 8 (Termination)**.** *If all correct replicas mba-propose, every correct replica eventually mba-decides some value.*

Proof: There are two cases considering the values correct replicas *mba-propose*: 1) at least $f + 1$ correct replicas *mba-propose* the same value v; 2) fewer than $f + 1$ correct replicas *mba-propose* the same value. In the following, we first prove that for the two cases, RABA_{id} eventually terminates, and then show that every replica eventually *mba-decides*.

Case 1) According to the protocol, all correct replicas that do not *mba-propose* v will eventually receive $f + 1$ $\mathsf{disperse}(v)$ and then broadcast $\mathsf{echo}(v)$. Every correct replicas will receive $n - f$ $\mathsf{disperse}(v)$ and $\mathsf{echo}(v)$, such that $rd[v] \geq n - f$. Then every correct replica broadcasts a $\mathsf{forward}(v, -)$ message. Similarly, every correct replica will eventually receive $n - f$ $\mathsf{forward}()$ messages and then *r-propose* some value to RABA_{id}. There are two sub-cases: *A*) At least one correct replica *r-proposes* 1 to RABA_{id}; *B*) None of the correct replicas *r-propose* 1 to RABA_{id}.

- *A*) From Lemma 7, every correct replica eventually *r-proposes* or *r-reproposes* 1. Hence, the biased termination property of RABA guarantees that RABA_{id} eventually terminates.
- *B*) If none of the correct replicas *r-reproposes* to RABA_{id}, the unanimous termination property of RABA guarantees RABA_{id} eventually terminates. If at least one correct replica *r-reproposes* 1, then according to Lemma 7, any correct replica will eventually *r-repropose* 1. The biased termination property of RABA guarantees that RABA_{id} eventually terminates.

Case 2) Fewer than $f + 1$ correct replicas *mba-propose* the same value. In this case, one of the following conditions is satisfied for any correct replica: 1) $\Sigma_x rd[x] - rd[v] \geq f+1$; 2) $\forall x \neq pv$, $\Sigma_x rd[x] \geq f+1$, i.e., every correct replica receives $f+1$ $\mathsf{disperse}(v)$ or $\mathsf{echo}(v)$ messages such that v is different from pv. The first condition holds if the correct replica receives messages from all replicas in the system, as given any value v, $rd[v] \leq f$, $\Sigma_x rd[x] \geq n-f$. The second condition holds as follows: considering the inputs of all correct replicas, for the value v for any correct replica, fewer than $f+1$ correct replicas *mba-propose* so more than $f+1$ correct replicas must *mba-propose* values different from v. Thus, every correct replica eventually receives $f+1$ $\mathsf{disperse}()$ messages such that the carried value is different from the replica's pv. After that, every correct replica that has not started RABA_{id} eventually *r-proposes* some value. If all correct replicas *r-propose* 0, RABA_{id} terminates according to the unanimous termination property of RABA. If at least one correct replica *r-proposes* 1, according to Lemma 7, every correct replica either *r-proposes* 1 or *r-reproposes* 1. RABA_{id} terminates according to the biased termination property.

In both cases, after RABA_{id} outputs, there are two cases: every correct replica *r-decides* 1; every correct replica *r-decides* 0. In the first case, according to Lemma 3, every correct replica either has already set rv as v or will eventually receive $\mathsf{distribute}(v, -)$. Then correct replica eventually *mba-decides*. In the second case, every correct replica *mba-decides* according to the protocol. ■

Theorem 9 (Integrity)**.** *Every correct replica mba-decides once.*

Proof: Every correct replica *mba-decides* after it *r-decides*. According to the integrity of RABA, every correct replica *r-decides* once so every correct replica *mba-decides* once. ■

Lemma 10. *Let nb be the maximal number of distinct values a correct replica may send in an* $\mathsf{echo}()$ *message. We have* $nb \leq 2$.

Proof: Every correct replica broadcasts an $\mathsf{echo}(v)$ message only if $v \neq pv$ and it has received $n-2f$ $\mathsf{disperse}(v)$. As there are n replicas in total, every correct replica sends a $\mathsf{echo}()$ message at most twice, i.e., $nb \geq 2$. ■

Theorem 11. *The time complexity of DO-MBA is expected* $O(1)$.

Proof: According to Lemma 10, $\mathsf{disperse}()$ and $\mathsf{echo}()$ runs in $O(1)$ time. Also, as each correct replica sends one $\mathsf{forward}()$ message and one $\mathsf{distribute}()$ message and RABA runs in $O(1)$ expected time. Thus, DO-MBA terminates in $O(1)$ expected time. ■

Theorem 12 (Total order)**.** *If a correct replica nd-delivers o before nd-delivering o′, then no correct replica nd-delivers o′ without first nd-delivering o.*

Proof: We assume that a correct replica p_i *nd-delivers* o with sequence number sn and *nd-delivers* o' with sn', where $sn < sn'$. We assume another correct replica p_j *nd-delivers* o with sn_1 and *nd-delivers* o' with sn_1', where $sn_1 > sn_1'$. We then prove the theorem by contradiction.

If p_i *nd-delivers* o with sequence number sn and p_j *nd-delivers* o with sn_1 such that $sn \neq sn_1$, p_j has *nd-delivered* a value $o'' \neq o$ with sequence number sn, as a correct replica never *nd-delivers* the same value twice. There are two cases: p_i *a-delivers* sn, o and p_j *a-delivers* sn, o'', a violation of the safety property of atomic broadcast; p_i and p_j both *a-deliver* sn, o, p_i *mba-decides* (v, v) and p_j *mba-decides* (v', v') such that $v' \neq v$, a violation of the primary agreement property of DO-MBA. Therefore, $sn_1 = sn$.

Similarly, if p_i *nd-delivers* o' with sn' and p_j *nd-delivers* o' with sn_1', $sn_1' = sn'$. We already know that $sn = sn_1$. Additionally, according to the assumption, $sn' > sn$ and $sn_1 > sn_1'$. Therefore, it holds that $sn' > sn_1'$, a contradiction with $sn_1' = sn'$. ■

Theorem 13 (Correctness)**.** *If a correct replica maintains state s before it nd-delivers o and maintains s′ after it nd-delivers o, another correct replica maintains state s before it nd-delivers o and maintains s″ after it nd-delivers o, then* $s' = s''$.

Proof: We consider that a correct replica p_i maintains s' at the end of the epoch (after DO-MBA outputs) and another correct replica p_j maintains s'' at the end of the epoch.

Let $s_1 \leftarrow \textit{execute}(s, o)$ be the execution result at p_i. There are three cases for s' at p_i: 1) DO-MBA outputs (v, h), where $v \neq \bot$ and $h \neq \bot$; 2) DO-MBA outputs (v, h), where $v \neq \bot$ and $h = \bot$; 3) DO-MBA outputs (v, h), where $v = \bot$. Similarly, the same three cases apply for p_j, considering $s_2 \leftarrow \textit{execute}(s, o)$. We prove that $s' = s''$ for each of the three cases for p_i.

- *Case 1)* In this case, according to the protocol, $s' = hash(s_1) = v$. According to the primary agreement and weak secondary agreement properties of DO-MBA, the output of p_j can only be v, h or $v, \bot$. If p_j *mba-decides* (v, h), we have $s'' = hash(s_2) = v$. Therefore, $s' = s''$. If p_j *mba-decides* $(v, \bot)$, we know that $hash(s_2) \neq v$. According to the protocol, p_j performs state transfer until $hash(s'') = v$. The non-intrusion property of DO-MBA ensures that at least one correct replicas holds s'', so p_j will complete the state transfer. Therefore, $s' = s''$.
- *Case 2)* In this case, we know that $hash(s_1) \neq v$ for p_i. According to the protocol, p_i performs state transfer until $hash(s') = v$. Furthermore, according to the primary agreement and weak secondary agreement properties of DO-MBA, the output of p_j can only be v, h or $v, \bot$. If p_j *mba-decides* (v, h), we know that $s'' = hash(s_2) = v$. Therefore, $s' = s''$. If p_j *mba-decides* $(v, \bot)$, we know that $hash(s_2) \neq v$. According to the protocol, p_j performs state transfer until $hash(s'') = v$. Therefore, $s' = s''$.
- *Case 3)* In this case, as p_i rolls back to the prior state, $s' = s$. According to the primary agreement property of DO-MBA, p_j must have *mba-decided* $(\bot, \bot)$ and rolled back to s. Thus, it holds $s' = s''$.

■

Theorem 14 (Liveness)**.** *If an operation o is submitted to all correct replicas, then each correct replica eventually nd-delivers o or ⊥; if o is deterministic, each correct replica nd-delivers o and updates its state via update.*

Proof: According to the liveness property of atomic broadcast, every correct replica will eventually *a-deliver*(sn, o). After *a-deliver*(sn, o), every correct replica queries $s_{sn} \leftarrow$ *execute*(s_{sn-1}, o) and starts a DO-MBA instance. According to the termination property of DO-MBA, every correct replica eventually *mba-decides* some value and then *nd-delivers* o or $\perp$.

We now prove that if o deterministic, every correct replica eventually *nd-delivers* o and updates it state via *update*. As s_0 is the same for all correct replicas, we prove that o will eventually be executed by an induction on sequence number. Without loss of generality, we consider each block consists of one transaction o. The case where each block consists of multiple transactions can be proved similarly.

For the base case, $sn = 1$. As s_0 is the same for all correct replicas and o consists of only deterministic operations, s_1 must be the same for all correct replicas. Therefore, all correct replicas *mba-propose* $v = hash(s_1)$. The validity property of DO-MBA guarantees that all correct replicas will *mba-decide* (v, v) and then *nd-deliver* o. The s_1 state includes *update* on o.

For the induction case, consider $sn > 1$ and all correct replicas maintain the same s_{sn-1}, we prove that all correct replicas will execute o and *a-deliver* o. In particular, as s_{sn-1} is the same for all correct replicas, all correct replicas will execute o and obtain the same state s_{sn}. All correct replicas will then *mba-propose* $v = hash(s_{sn})$ and *mba-decide* (v, v). Every correct replica then *nd-delivers* o and the state s_{sn} includes the *update* on o. ■