Ipotane: Balancing the Good and Bad Cases of Asynchronous BFT

Xiaohai Dai*, Chaozheng Ding*, Hai Jin*, Julian Loss†, and Ling Ren‡
*National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology
†CISPA Helmholtz Center for Information Security
‡University of Illinois at Urbana-Champaign
{xhdai, chaozhengding, hjin}@hust.edu.cn, loss@cispa.de, renling@illinois.edu

***Abstract*—State-of-the-art asynchronous *Byzantine Fault Tolerance* (BFT) protocols integrate a partially-synchronous optimistic path. Their ultimate goal is to match the performance of a partially-synchronous protocol in favorable situations and that of a purely asynchronous protocol in unfavorable situations. While prior works have excelled in favorable situations, they fall short when conditions are unfavorable. To address these shortcomings, a recent work, Abraxas (CCS'23), retains stable throughput in all situations but incurs very high worst-case latency in unfavorable situations due to slow detection of optimistic path failures. Another recent work, ParBFT (CCS'23) ensures good latency in all situations but suffers from reduced throughput in unfavorable situations due to the use of extra *Asynchronous Binary Agreement* (ABA) instances.**

We propose Ipotane, a protocol that attains performance comparable to partially-synchronous protocols in favorable situations and to purely asynchronous ones in unfavorable situations, in terms of both throughput *and* latency. Ipotane also runs two paths simultaneously: 2-chain HotStuff as the optimistic path and a new primitive *Dual-functional Byzantine Agreement* (DBA) for the pessimistic path. DBA packs the functionalities of biased ABA and *Validated Asynchronous Byzantine Agreement* (VABA). In Ipotane, each replica inputs 0 to DBA if its optimistic path is faster, and 1 if its pessimistic path is faster. DBA's ABA functionality promptly signals the optimistic path's failure by outputting 1, ensuring Ipotane's low latency in unfavorable situations. Meanwhile, Ipotane executes DBA instances to continuously produce pessimistic blocks through their VABA functionality. Upon detecting a failure, Ipotane commits the last two pessimistic blocks to maintain high throughput. Moreover, Ipotane leverages DBA's biased property to ensure the safety of committing pessimistic blocks. Extensive experiments validate Ipotane's high throughput and low latency across all situations.

I. Introduction

The explosive popularity of blockchain technology [1], [2] and Web3 ecosystem [3], [4] has reignited significant interest in *Byzantine Fault Tolerant* (BFT) consensus over the past decade [5], [6]. At its core, BFT consensus allows distributed replicas to reach agreement even in scenarios where a subset of these replicas, termed Byzantine replicas, may deviate arbitrarily from the protocol.

BFT consensus protocols traditionally fall into three categories based on their network assumptions: asynchronous, partially synchronous, and synchronous. Asynchronous protocols [7], [8], [9] ensure safety and liveness under arbitrary network conditions, whereas (partially-)synchronous protocols are prone to network attacks [10]. On the flip side, asynchronous protocols are known to be inherently randomized [11], which makes them less efficient and more challenging to design than their synchronous and partially-synchronous counterparts (e.g., PBFT [12] and HotStuff [13]), which can be fully deterministic.

A. Asynchronous protocols with an optimistic path

To harness the strengths of both (partially-)synchronous and asynchronous protocols, a line of research has proposed incorporating an optimistic path into an asynchronous protocol [14], [15], [16], [17]. This typically involves using a partially-synchronous protocol, like 2-chain HotStuff [18], as the optimistic path, while an asynchronous protocol, often *Validated Asynchronous Byzantine Agreement* (VABA), acts as the pessimistic fall-back path.

This dual-path paradigm considers two situations: favorable and unfavorable. A favorable situation is characterized by a non-faulty leader on the optimistic path and good network conditions, enabling the protocol to make progress through the optimistic path. In contrast, an unfavorable situation arises when we have a faulty leader or poor network conditions, in which case the protocol will fall back to the pessimistic path to achieve liveness. Formal definitions of favorable/unfavorable situations are presented in Section V-D.

The ultimate goal in this dual-path paradigm is to match the performance of a partially-synchronous protocol in favorable situations and that of a purely asynchronous protocol in unfavorable ones. While many prior works have focused on optimizing performance in favorable situations, it is also critical to address performance in unfavorable situations, as they can be common in real-world deployments. Specifically, a leader may become temporarily inoperative, or the network connecting to the leader might experience jitter, making such

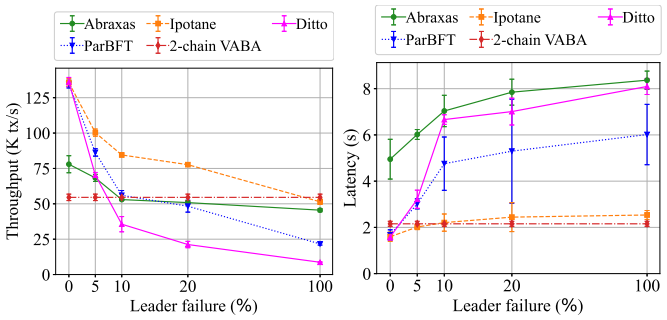
Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230003
www.ndss-symposium.org

TABLE I: Performance comparison. δ and Δ denote the actual network delay and timer parameter. c represents the maximum transaction count of a block, while λ is the lookback parameter in Abraxas. Performance in unfavorable situations holds even when the adversary mounts arbitrary attacks.

	Favorable situations		Unfavorable situations	
	Latency	Throughput	Latency	Throughput
2-chain HotStuff [14]	5δ	$c/(2\delta)$	/	/
2-chain VABA [14]	10.5δ	$2c/(7\delta)$	10.5δ	$c/(7\delta)$
Ditto [14]	5δ	$c/(2\delta)$	$3\Delta + 10.5\delta$	$c/(2\Delta + 7\delta)$
Abraxas [17]	5δ	$c/(2\delta)$	$3.5\lambda\delta + 14\delta^{\dagger}$	$c/(7\delta)$
ParBFT [16]	5δ	$c/(2\delta)$	22δ	$c/(22\delta)$
Ipotane	5δ	$c/(2\delta)$	18.5δ	$3c/(23\delta)$

[§]Latency here refers to consensus latency—measured from block generation to being committed—rather than end-to-end latency, following almost all works in the literature. This distinction is made because end-to-end latency includes not only consensus latency but also mempool queuing time, which is hard to quantify analytically.

[†] λ cannot be set too small, as this would make Abraxas resort to pessimistic paths too often, degrading performance. The Abraxas paper recommends setting λ to 20 [17].



(a) Throughput comparison (b) Latency comparison

Fig. 1: Performance under varying probabilities (ρ) of leader failure. ρ=0 denotes a favorable situation where optimistic paths operate smoothly. Conversely, ρ=100% denotes an unfavorable situation where optimistic paths fail.

situations frequently occur. Furthermore, even a short period of poor consensus performance, particularly those resulting from unfavorable situations, can significantly degrade user experience in upper-layer applications and should be diligently avoided.

While existing protocols successfully achieve high performance in favorable situations, a significant gap remains in unfavorable situations. Specifically, earlier works like Ditto [14] and BDT [15] follow a *sequential-path* design where the pessimistic path is launched only after the optimistic path's failure is detected. This delay in launching the pessimistic path results in poor efficiency in unfavorable situations. We give a more thorough comparison with these and other existing works in Section VII.

To deal with issues of sequential-path protocols, two recent works ParBFT [16][1] and Abraxas [17], follow a *parallel-path* design which operates two paths simultaneously. By continuously running the pessimistic path in the background, parallel-path protocols avoid much of the overhead encountered in the sequential-path design during unfavorable situations.

In spite of these improvements, ParBFT and Abraxas still fall short of fully matching the performance of asynchronous protocols in unfavorable situations. Concretely, ParBFT employs an individual *Asynchronous Binary Agreement* (ABA) instance at each height to detect the optimistic path's failure. This achieves low latency in unfavorable situations but introduces an idle period where no new block is generated, resulting in reduced throughput (i.e., number of committed blocks) compared to purely asynchronous protocols. On the other hand, Abraxas's pessimistic path leverages consecutive VABA instances to continuously generate blocks even during optimistic periods. Since there is no idle time, Abraxas achieves essentially the same throughput as purely asynchronous protocols. The downside, however, is that blocks from the pessimistic path can be committed only after an indicator transaction on the pessimistic path confirms the optimistic path's failure. This indicator transaction is submitted after many communication rounds (empirically, over 80 rounds). Thus, Abraxas might incur very high latency in the worst case.

Therefore, we aim to design an asynchronous protocol whose throughput and latency are on par with the state-of-the-art of: 1) partially-synchronous protocols under favorable situations, and 2) purely asynchronous protocols in unfavorable situations.

B. Our solution

We answer this question affirmatively by proposing Ipotane. Ipotane combines the advantages of Abraxas and ParBFT. More precisely, Ipotane delivers performance akin to that of partially-synchronous protocols under favorable situations; in unfavorable situations, Ipotane still offers throughput and latency comparable to those of a purely asynchronous protocol. In addition, in situations that fall between favorable and unfavorable, Ipotane consistently maintains nearly the best throughput and latency among existing protocols.

At a high level, Ipotane executes the optimistic and pessimistic paths in parallel, much like Abraxas and ParBFT. The optimistic path runs the 2-chain HotStuff, whereas the pessimistic path involves a sequence of asynchronous consensus instances. These instances have dual functions: they monitor whether the optimistic path works well like ABA, and also facilitate the generation of new blocks reminiscent of VABA. In addition, the 2-chain HotStuff's committing rule only ensures that $t+1$ non-faulty replicas acquire the lock data, which is then taken as the input to the asynchronous instance. Inputs from these $t+1$ non-faulty replicas must force the asynchronous instance to produce a matching output, calling for a *biased validity*.

We introduce *Dual-functional Byzantine Agreement* (DBA), a novel primitive to implement the asynchronous instance discussed above, which combines the functionalities of biased ABA and VABA. In addition to the validated block, as required

[1]In [16], two versions of ParBFT are proposed. Our focus is on the first one, ParBFT1, which we simply refer to as ParBFT in this paper.

by standard VABA, the input for a DBA instance also includes a binary value. The output is a pair comprising a binary value and a block value, particularly ensuring biased validity for the binary value. DBA can be constructed by adding merely a single communication round prior to any existing VABA protocol. Thus, its performance is similar to VABA.

With the DBA primitive defined, Ipotane executes consecutive DBA instances as the pessimistic path. The binary decision from DBA indicates the success or failure of the optimistic path. Upon detecting a failure in the optimistic path, Ipotane promptly commits blocks on the pessimistic path, thus promising low latency in unfavorable situations. Due to the biased validity of DBA, if any non-faulty replica commits a block through the optimistic path, the binary output from DBA acknowledges this commit. This also instructs every non-faulty replica to commit the same block through the pessimistic path if it has not yet committed, thus guaranteeing the block consistency. Moreover, DBA instances are comparable in efficiency to VABA, which helps Ipotane achieve good throughput under unfavorable situations.

Experimental results to evaluate Ipotane are shown in Figure 1, where the x-axis represents the probability (ρ) of leader failure on the optimistic path. It shows that under favorable situations ($\rho = 0$), Ipotane achieves high throughput and low latency, matching Ditto, which operates as a purely partially-synchronous protocol in such situations. On the other hand, when leaders are always faulty ($\rho = 100\%$), Ipotane demonstrates throughput and latency on par with 2-chain VABA, a purely asynchronous protocol. Furthermore, as the probability of leader failure varies between 0 and 100%, reflecting a mix of favorable and unfavorable situations occurring randomly, Ipotane consistently achieves almost the highest throughput and lowest latency compared to other protocols.

Table I presents a more detailed and comprehensive comparison between Ipotane and existing protocols, corroborating Ipotane's good performance with theoretical analysis. δ denotes the actual network delay, and c represents the maximum transaction count within a block. Ipotane demonstrates a low latency of 5δ and a high throughput of $c/(2\delta)$ in favorable situations. This matches the performance of a partially-synchronous protocol (specifically, 2-chain HotStuff). On the other hand, in unfavorable situations, Ipotane manages to maintain a latency of 18.5δ[2] and a throughput of $3c/(23\delta)$. These are just slightly worse than a purely asynchronous protocol (specifically, 2-chain VABA) but significantly better than prior works Abraxas in terms of latency and ParBFT in terms of throughput.

II. Models and Preliminaries

In this section, we begin by introducing the system model. Building upon this model, we formally define *State Machine Replication* (SMR), which is primarily achieved through the use of asynchronous BFT consensus. Finally, we present two preliminary protocols, *Validated Asynchronous Byzantine Agreement* (VABA) and (biased) *Asynchronous Binary Agreement* (ABA), which serve as a basis for our subsequent design.

A. Model

The system consists of n replicas, with up to t being Byzantine, where $n \geq 3t + 1$. Each replica is identified by a unique number and is denoted as p_i ($1 \leq i \leq n$). Byzantine replicas may deviate from the protocol arbitrarily and are presumed to be under the control of an adaptive adversary. This adversary can corrupt replicas as the protocol progresses and drop a corrupted replica's messages from the network a posteriori. The remaining replicas, termed non-faulty, faithfully adhere to the protocol. Each pair of replicas is connected through a pairwise authenticated communication channel. The system operates in an asynchronous network where no assumption is made about network delays. The adversary is assumed to fully control the network and can arbitrarily delay and reorder any messages as long as it eventually delivers them.

A *Public Key Infrastructure* (PKI) is established across the replicas, and digital signatures are used to ensure the authenticity and integrity of transmitted messages. Additionally, we employ two distinct instances of threshold signature schemes [19], [20]: one with a threshold of $n - t$, and the other with a threshold of $t+1$. The algorithm for generating a threshold signature share is denoted as $\mathsf{SignShr}$, while Comb constructs a threshold signature from sufficient shares. To simplify our notation, we omit the use of private or public keys as parameters in $\mathsf{SignShr}$ or Comb. To differentiate between the two threshold signature schemes, we use $\mathsf{SignShr}_r$ and Comb_r to denote calls to these algorithms with the threshold parameter r. We assume the adversary is computationally bounded and cannot break the security of (threshold) signatures.

B. State machine replication

We focus on the *State Machine Replication* (SMR) problem. Each replica p_i in SMR locally maintains a growing chain, denoted as $\mathcal{C}_i$, which is modeled as a write-once array. An object in the array is named a block, which consists of multiple transactions. Transactions are continuously generated by clients or upper-layer applications, and are inserted into a buffer buf_i of each replica i. Transactions cached in the buffer are sorted based on the times they are received by the replica. When a replica p_i proposes a block, it selects a number of transactions from its buffer buf_i. Without loss of generality, we assume the maximum number of transactions that can be included in a block is c. Therefore, the block proposed by p_i consists of the first c transactions from the buffer $\mathsf{buf}_i[:c]$.

$\mathcal{C}_i$ is initialized as empty—namely $\mathcal{C}_i[k] = \perp$ for each index k ($k \geq 1$). A block B is said to be committed by p_i when it is written to the chain $\mathcal{C}_i$. All transactions in B are then deleted from p_i's buffer buf_i. In this paper, we focus on protocols that commit blocks sequentially—i.e., if $\mathcal{C}_i[k] \neq \perp$, then for every $k' < k$, $\mathcal{C}_i[k'] \neq \perp$. SMR serves to maintain a consistent chain among non-faulty replicas, whose definition is as follows:

[2]All unfavorable latency metrics are measured in terms of expected and average values. Specifically, we first compute the expected latency for each block and then take the average across multiple blocks.

Definition 1. *Let* Π *be a protocol executed among replicas* $p_1, ..., p_n$*, where each non-faulty replica holds a transaction buffer* buf_i*. We say that* Π *implements SMR if it satisfies the following properties:*

- ***Consistency:*** *For two non-faulty replicas* p_i *and* p_j*, if* $\mathcal{C}_i[k] \neq \perp$ *and* $\mathcal{C}_j[k] \neq \perp$*, then* $\mathcal{C}_i[k] = \mathcal{C}_j[k]$*.*
- ***Liveness:*** *If a transaction* tx *is added to every non-faulty replica's buffer, then every non-faulty replica will eventually commit a block containing* tx*.*
- ***Completeness:*** *For two non-faulty replicas* p_i *and* p_j*, if* $\mathcal{C}_i[k] \neq \perp$*, then for each index* $k' \leq k$*, eventually* $\mathcal{C}_j[k'] \neq \perp$*.*

C. Validated asynchronous byzantine agreement

The *Validated Asynchronous Byzantine Agreement* (VABA) abstraction facilitates consensus on arbitrary values [21]. VABA introduces an external validation predicate Q, typically defined by higher-layer applications. To be more specific, VABA is defined as follows.

Definition 2. *Let* Π *be a protocol executed among replicas* $p_1, ..., p_n$*, where replica* p_i *holds input* v_i *and replicas terminate upon generating output. We say that* Π *achieves VABA if it satisfies the following properties in an asynchronous network whenever at most* t *replicas are corrupt:*

- ***Agreement:*** *If two non-faulty replicas output values* v *and* v'*, then* $v = v'$*.*
- ***Termination:*** *If all non-faulty replicas complete inputting, every non-faulty replica will eventually output.*
- ***External validity:*** *If a non-faulty replica outputs* v*, then* $Q(v)$ *must be* True*.*
- ***Quality:*** *If a non-faulty replica outputs* v*, then with probability over* $1/2$*,* v *is input by a non-faulty replica.*

Various implementations of VABA [21], [22], [23], [24] have been developed over the past decades. While the quality property is not explicitly defined in [21], [24], both works guarantee it.

D. (Biased) asynchronous binary agreement

The *Asynchronous Binary Agreement* (ABA) abstraction [25], [26] represents the most basic form of asynchronous BFT consensus, which serves to agree on a binary value. To be more specific, an ABA protocol is defined as follows:

Definition 3. *Let* Π *be a protocol executed among replicas* $p_1, ..., p_n$*, where replica* p_i *holds a binary input* b_i *and generates an output. We say that* Π *achieves ABA if it satisfies the following properties in an asynchronous network whenever, at most* t *replicas are corrupt:*

- ***Agreement:*** *If two non-faulty replicas output values* b *and* b'*, then* $b = b'$*.*
- ***Termination:*** *If all non-faulty replicas complete inputting, every non-faulty replica will eventually output.*
- ***Validity:*** *If all non-faulty replicas input the same bit* b*, then each non-faulty replica outputs* b*.*

In the VABA implementation proposed by Cachin et al. [21], a binary version of the VABA protocol with a biased validity property is introduced. This protocol closely resembles the ABA protocol, and for presentation, we refer to it as biased ABA. Biased ABA replaces the validity property in the ABA protocol with external validity and biased validity properties. The external validity property remains consistent with the definition in VABA, where a validation predicate P is defined over binary values, while the biased validity property is defined as follows:

- ***Biased validity:*** If at least $t + 1$ non-faulty replicas input the bit 0, all non-faulty replicas will output 0.

Our design does not utilize a biased ABA directly. Instead, we introduce a new abstraction named DBA that incorporates properties akin to those in biased ABA, which is detailed in Section III.

III. Building Block: DBA

A. Definition of DBA

We propose a new abstraction called *Dual-functional Byzantine Agreement* (DBA), which simultaneously achieves consensus on a binary value as well as an arbitrary value. Roughly speaking, DBA combines the functionalities of biased ABA and VABA. Initially, it may seem that VABA inherently fulfills the functionality of biased ABA, making the definition of DBA redundant. However, this is not the case, as biased ABA has a variant of the validity property (i.e., biased validity) from VABA. In the context of this paper, the arbitrary value is typically a block. Therefore, within the remainder of this paper, we will use the term *block* to represent the arbitrary value in DBA. Formally, a DBA protocol is defined as follows:

Definition 4. *Let* Π *be a protocol executed among replicas* $p_1, ..., p_n$*, where replica* p_i *holds a binary value* b_i*, a proof* σ*, plus a block* B_i *as input, and generates a binary value* b *and a block* B *as output. Two external validation predicates,* P *and* Q*, are introduced:* Q *validates the legitimacy of the block value, similar to the validation in VABA, while* P *validates the legitimacy of the binary value based on the proof. Replicas terminate upon generating output. We say that* Π *achieves DBA if it satisfies following properties whenever at most* t *replicas are corrupt:*

- ***Agreement:*** *For any two non-faulty replicas outputting* $\langle b, B\rangle$ *and* $\langle b', B'\rangle$*, then* $b = b'$ *and* $B = B'$*.*
- ***Termination:*** *If all non-faulty replicas complete inputting, every non-faulty replica will eventually output.*
- ***Block-external validity:*** *For any output* $\langle *, B\rangle$ *from a non-faulty replica,* $Q(B) = \mathsf{True}$*.*
- ***Quality:*** *If a non-faulty replica outputs* $\langle *, B\rangle$*, then with probability over* $1/2$*,* B *is input by a non-faulty replica.*
- ***Biased validity.*** *If at least* $t + 1$ *non-faulty replicas input* $\langle 0, *\rangle$*, then all non-faulty replicas will output* $\langle 0, *\rangle$*.*

Algorithm 1: AlgDBA with instance identity h (for p_i)

1 **Let** $\langle b_i, \sigma_i, B_i \rangle$ denote the input of p_i.
2 **if** $b_i = 0$ **then**
3 | **broadcast** $(h, 0, \sigma_i, \mathsf{SignShr}_{t+1}(h, 0))$
4 **else**
5 | **broadcast** $(h, 1, \sigma_i, \mathsf{SignShr}_{n-t}(h, 1))$
6 **on** *receiving* $(h, 0, \sigma_j, *)$ *s.t.* $P(\sigma_j) = \mathsf{True}$ *from* p_j**:**
| // amplify the bit of 0
7 | **if** p_i *has not broadcast* 0 **then**
8 | | **broadcast** $(h, 0, \sigma_j, \mathsf{SignShr}_{t+1}(h, 0))$
9 **on** *receiving* $t+1$ $(h, 0, \sigma_j, *)$ *that* $P(\sigma_j) = \mathsf{True}$**:**
10 | $S \leftarrow$ all the sig. shares from $t+1$ messages;
11 | $sig_0 \leftarrow \mathsf{Comb}_{t+1}(h, 0, S)$;
12 | **input** $\langle 0, sig_0, B_i \rangle$ to VABA_h **if** it has not input
13 **on** *receiving* $n-t$ $(h, 1, *, *)$**:**
14 | $S \leftarrow$ all the sig. shares from $n-t$ messages;
15 | $sig_1 \leftarrow \mathsf{Comb}_{n-t}(h, 1, S)$;
16 | **input** $\langle 1, sig_1, B_i \rangle$ to VABA_h **if** it has not input
17 **on** *outputting* $\langle b, sig, B \rangle$ *from* VABA_h**:**
18 | **output** $\langle b, B \rangle$

- ***Binary-external validity:*** *If a non-faulty replica outputs* $\langle 0, * \rangle$*, at least one replica (Byzantine or non-faulty) must have inputted* $\langle 0, \sigma, * \rangle$ *with* $P(\sigma) = \mathsf{True}$.

To aid presentation, in the context of DBA's input, b plus σ is referred to as the *binary input*, while B is termed the *block input*. Correspondingly, in the output $\langle b, B \rangle$, we refer to b and B as the *binary output* and *block output*, respectively. The block inputs of different replicas, like the input values in VABA, do not have to be identical. The properties of biased validity and binary-external validity specifically pertain to DBA's binary values, while quality and block-external validity are applicable to the block values.

B. Construction of DBA: AlgDBA

Since the combination of binary input and block input can be regarded as a single value, an initial approach to constructing DBA might involve adapting a VABA protocol to accept the combined inputs as a singular value. This approach could fulfill most of the properties outlined in Definition 4, but it falls short of meeting the biased validity requirement. To solve this problem, we introduce a communication round before executing the VABA protocol.

This protocol, as outlined in Algorithm 1, adds a round to amplify the bit of 0 before executing VABA. Unless otherwise specified, the term *broadcast* in the pseudocode refers to a *best-effort broadcast* [27], a simple form of broadcast where the broadcaster transmits data to other replicas, and these replicas immediately deliver the data upon receipt. In this additional round, each replica broadcasts its binary input accompanied by a threshold signature share. If the binary input is 0, the threshold parameter for the signature share is set to $t+1$. Conversely, for a binary input of 1, the threshold is set to $n-t$ (see Lines 2-5). If a replica receives a valid message containing 0, it will also broadcast 0 if it has not yet done so (Lines 6-8), which amplifies the broadcast of 0.

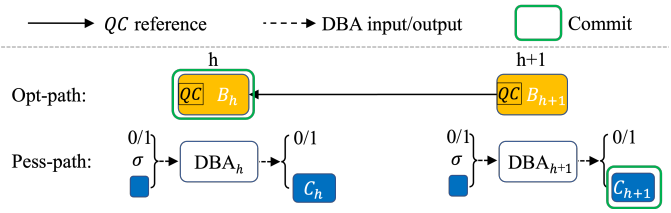


Fig. 2: The structure of an epoch in Ipotane

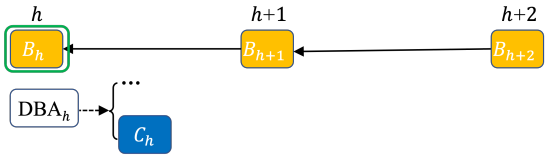
At the end of this round, if a replica gathers $t+1$ messages containing 0, it creates a complete threshold signature sig_0 based on signature shares in these messages, certifying the bit 0. The replica then uses $\langle 0, sig_0, B \rangle$ as input to the VABA instance, where B is the block input of AlgDBA (Lines 9-12). Alternatively, if it receives $n-t$ valid values of 1, it creates a complete threshold signature sig_1 for the value 1, leading to the input $\langle 1, sig_1, B \rangle$ for the following VABA instance (Lines 13-16). Finally, VABA outputs one of these inputs, with the bit and block values forming the output of AlgDBA (Lines 17-18). Within this construction, the predicate Q of VABA must validate both the binary and block parts of an input. In particular, validation of the binary part typically involves verifying the bit's threshold signature. For lack of space, the correctness analysis of AlgDBA is deferred to Appendix A.

Relation to Cachin et al. [21]. Cachin et al. [21] first introduced VABA and its variant, biased ABA. While we draw inspiration from their work, our work is significantly different from theirs. The end goal of Cachin et al. [21] is to construct VABA that agrees on a block value, and they define and use the biased ABA in that process. In contrast, our DBA is a new primitive that simultaneously achieves agreement on a binary value and a block value. To obtain DBA, we modify existing VABA protocols and use them as building blocks.

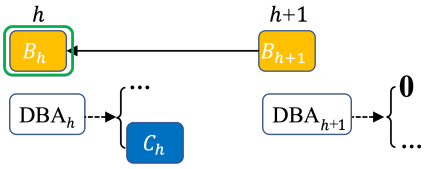
IV. Ipotane Design

A. Overview and intuition

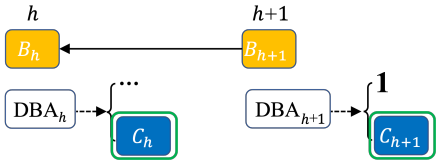
Ipotane operates in epochs, designated by incrementing integer identifiers starting from 1. Each epoch comprises an optimistic path and a pessimistic path in parallel, as depicted in Figure 2. The optimistic path employs a structure of chain-based blocks, where the *Quorum Certificate* (QC) for a block is encapsulated within the next block. The pessimistic path is implemented through consecutive DBA instances, each producing a block and a binary value. Blocks generated in the two paths are referred to as *opt-blocks* and *pess-blocks*, respectively. Opt-blocks within an epoch are numbered with heights starting from 1, denoted as B_h. Similar to a partially-synchronous protocol, a leader is designated for each height on the optimistic path, following a round-robin manner. DBA instances and their outputted pess-blocks in an epoch are also numbered starting from 1, denoted as DBA_h and C_h, respectively.



(a) Commit B_h through the two-chain rule.



(b) DBA$_{h+1}$ outputs 0 indicating readiness to commit B_h.



(c) DBA$_{h+1}$ outputs 1 indicating readiness to commit C_h and C_{h+1}.

Fig. 3: Examples to show the block committing rules. We omit some elements in the figures for conciseness.

1) Design intuition: In the context of Ipotane, we make a clear distinction between the terms *certify* and *commit* concerning a block. An opt-block is deemed *certified* when the corresponding QC is obtained, and a pess-block is considered *certified* if it is outputted from a DBA instance. Taking Figure 2 as an example, the opt-block B_h is certified, since the QC for it is contained in B_{h+1}. Both pess-blocks C_h and C_{h+1} are certified, as they are outputted from DBA$_h$ and DBA$_{h+1}$, respectively. Due to the quorum intersection argument, the opt-block at a given height will be unique. Additionally, according to DBA's consistency property, the pess-block at a given height will also be unique. Conversely, *commit* denotes that a block, either an opt-block or a pess-block, is eligible to be written to the SMR chain $\mathcal{C}$.

Within this parallel-path structure, it is possible to have two certified blocks at the same height, h: an opt-block B_h and a pess-block C_h. A primary task is to decide which block to commit. This is precisely the reason why we augment VABA to DBA to make a binary decision. In particular, we leverage the binary output from the DBA instance **at the next height** DBA$_{h+1}$, to commit the block at height h. On the other hand, to attain performance comparable to a partially-synchronous protocol in favorable situations, Ipotane must be capable of rapidly committing blocks through the optimistic path, particularly employing the two-chain rule akin to 2-chain HotStuff [14]. Thus, two distinct rules for block committing co-exist: one using binary outputs on the pessimistic path, and the other using the two-chain rule on the optimistic path.

The next challenge is to ensure consistency between these two commit rules. Specifically, for a given height, if a replica commits an opt-block using the two-chain rule, we must ensure that another replica will also commit this opt-block even if it follows the pessimistic path's binary output. This consistency is achieved through the biased-validity property of DBA. In short, if a replica commits B_h via the two-chain rule, then at least $t+1$ non-faulty replicas have inputted 0 to DBA$_{h+1}$, signaling their intention to commit B_h. Due to the biased-validity property, DBA$_{h+1}$ will output 0, which indicates committing B_h.

2) Overall design: Each replica participates in both the optimistic and pessimistic paths. The optimistic path, resembling the 2-chain HotStuff, involves designated leaders proposing opt-blocks, which are then voted on by replicas using threshold signature shares. The pessimistic path, on the other hand, consists of consecutive DBA instances. The input for a DBA instance (DBA$_{h+1}$) depends on which block at the preceding height h—either opt-block B_h or pess-block C_h—gets certified first. An opt-block is certified by a QC contained in the subsequent opt-block, whereas a pess-block is certified upon being outputted from DBA. Thus, a replica's binary input for DBA$_{h+1}$ hinges on which of these two events occurs first: (1) receipt of B_{h+1} or (2) output from DBA$_h$. If B_{h+1} is received earlier, it inputs 0 to DBA$_{h+1}$; otherwise, it inputs 1.

Committing an opt-block or a pess-block is based on either the two-chain rule or the output from DBA. As depicted in Figure 3a, upon receiving an opt-block B_{h+2}, a replica can immediately commit the opt-block from two heights prior (B_h) via the two-chain rule. On the other hand, if a replica receives 0 from DBA$_{h+1}$, as illustrated in Figure 3b, it can commit the opt-block **at the preceding height** B_h. Otherwise (namely if DBA$_{h+1}$ outputs 1), the replica commits the pess-block **at the preceding height** C_h, as shown in Figure 3c.

In addition, the 1 output from DBA$_{h+1}$ indicates a failure in the optimistic path. In this scenario, each replica concludes the current epoch and progresses to the next. To enhance throughput, the pess-block C_{h+1} generated from DBA$_{h+1}$ is also committed together with C_h. As demonstrated in Figure 3c, both C_h and C_{h+1} are committed when DBA$_{h+1}$ outputs 1.

In favorable situations, Ipotane continuously commits blocks through the two-chain rule, achieving performance akin to partially-synchronous protocols. In contrast, under unfavorable situations, Ipotane can commit blocks using the pessimistic path, thereby ensuring liveness. Since DBA can be effectively constructed based on a VABA protocol with efficient modifications, DBA offers performance comparable to VABA, enabling Ipotane to match the performance of purely asynchronous protocols in unfavorable situations.

B. Data structures and utilities

We describe data structures and utilities in this section, which are summarized as Algorithm 2. An opt-block B_h on the optimistic path is characterized by the data structure $\{h, QC, d\}$, where h represents its height number, QC is a certificate for the preceding block B_{h-1}, and d denotes a transaction batch from the buffer buf.

On the pessimistic path, each replica can generate a transaction batch at a height h, serving as the block input to the

Algorithm 2: Data structures & utilities for p_i

```
1  struct Opt-Block:
2  |  {h, QC, d}
3  struct DBAInput:
4  |  {b, σ, C}
5  struct DBAOutput:
6  |  {b, C}
7  define GenOptBlk(h, QC):
8  |  d ← GenTxBatch();
9  |  B.h ← h; B.QC ← QC; B.d ← d;
10 |  return B
11 define InvokeDBA(h, b, σ):
12 |  d ← GenTxBatch();
13 |  I.b ← b; I.σ ← σ; I.C ← d;
14 |  invoke DBA_h with I
15 define GenTxBatch():
16 |  d ← a batch of transactions from buf_i;
17 |  return d
18 define Commit(blk):
19 |  len ← C_i.len(); C_i[len + 1] ← blk;
20 |  delete tx from buf_i for each tx ∈ blk
```

DBA_h instance. From these block inputs, only one is outputted from DBA_h and is referred to as *certified*, denoted as C_h. Consequently, a replica's input I to the DBA_h instance follows the format $\{b, \sigma, C\}$, where b is a binary value indicating its opinion on which block at height $h-1$ is certified earlier, and C denotes the block input. If $b = 0$, the replica believes the opt-block B_{h-1} is certified earlier, and σ is set to QC of B_{h-1}. Otherwise ($b = 1$), the replica believes that the pess-block C_{h-1} is certified earlier, leaving $\sigma = \perp$. The output from DBA_h is consistent across replicas, and has the format $\{b, C\}$, where b is a bit indicating the agreed-upon result regarding which block at height $h-1$ is certified earlier. C is a block output derived from one of the block inputs. For convenience, we omit the height numbers in the data structures of DBA inputs and outputs. Instead, their heights are implied by the height numbers of DBA instances. For example, "invoking DBA_h with I" implies I has a height h, and "DBA_h outputs O" implies O has a height h.

We also define some utilities for Ipotane, including $GenOptBlk$, $InvokeDBA$, and $Commit$. Both $GenOptBlk$ and $Invoke\text{-}DBA$ need to extract a batch of transactions from the replica's transaction buffer buf, which is achieved by calling the $GenTxBatch$ function.

C. Detailed design when $h > 1$

Algorithm 3 outlines an epoch in Ipotane, which operates in consecutive heights[3]. This subsection describes the general protocol for heights greater than 1, with special considerations for the first height discussed in the next subsection.

[3]We put termination and invocation of DBA (Lines 17-18 of Algorithm 3) as part of the optimistic path, as these actions are triggered by receiving an opt-block. Similarly, we put committing an opt-block (Lines 24-25) as part of the pessimistic path, as these actions are triggered by receiving a pess-block.

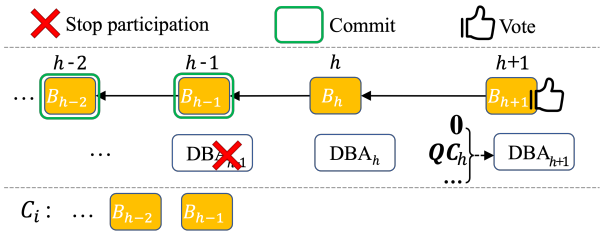


Fig. 4: Actions taken when $t_1 < t_2$ in Ipotane

The binary-external validity function P in DBA is defined as a $(n-t)$-threshold signature verification function. The binary input or output in the DBA instance is 0 if the corresponding opt-block is certified earlier than the pess-block, and is 1 otherwise. In other words, a replica inputs 0 if it believes the optimistic path is functioning well and inputs 1 if it perceives a lack of progress with the optimistic path. An output of 0 from a DBA instance indicates agreement among replicas that the optimistic path performs well, while an output of 1 indicates agreement that the optimistic path has encountered a failure.

For a height, consider two time points for replica p_i:

- t_1**:** the time when the opt-block B_h is certified, indicated by receiving an opt-block B_{h+1}
- t_2**:** the time when the pess-block C_h is certified, indicated by receiving the output from the DBA_h instance.

If the optimistic path operates effectively, the DBA instance DBA_h will be launched upon the reception of the opt-block B_h. Subsequently, it takes 2δ for B_h to be certified by the QC contained in the subsequent opt-block B_{h+1}, whereas a minimum of 7δ is required for DBA_h to output the certified pess-block C_h. Therefore, we should have $t_1 < t_2$ when the optimistic path is functioning well. The comparison between t_1 and t_2, hence, serves as an indicator of whether the optimistic path is working well. Replica p_i takes different actions based on this comparison.

1) Case 1: $t_1 < t_2$: In this case, p_i receives B_{h+1} earlier than the output from DBA_h, indicating that the optimistic path works well as expected. Replica p_i leverages the two-chain rule to commit block B_{h-1} (when $h \geq 2$) and casts a vote for the received block B_{h+1}, as described in Figure 4 and Lines 15-16 in Algorithm 3. Furthermore, p_i stops participating in the DBA_{h-1} (when $h \geq 2$) instance (line 17). In addition, p_i inputs to DBA_{h+1} its opinion that B_h is certified earlier than C_h. To be concrete, its binary input to DBA_{h+1} is 0 plus QC of B_h contained in B_{h+1} (Line 18). We denote QC of B_h as QC_h. This ensures the consistency of committed blocks. Intuitively, if a non-faulty replica commits B_h after receiving B_{h+2}, at least $t+1$ non-faulty replicas must have received B_{h+1} earlier and inputted 0 to DBA_{h+1}. Therefore, DBA's biased validity guarantees that DBA_{h+1} will output 0, directing any non-faulty replica to commit B_h if it has not done so already. Moreover, p_i will also broadcast B_{h+1} to make sure other replicas receive this block (Line 19).

Algorithm 3: An epoch in Ipotane for p_i

1 **Let** L_h denote the leader of height h on opt. path.
2 $h \leftarrow 1$, $prevPessBlk \leftarrow \perp$.

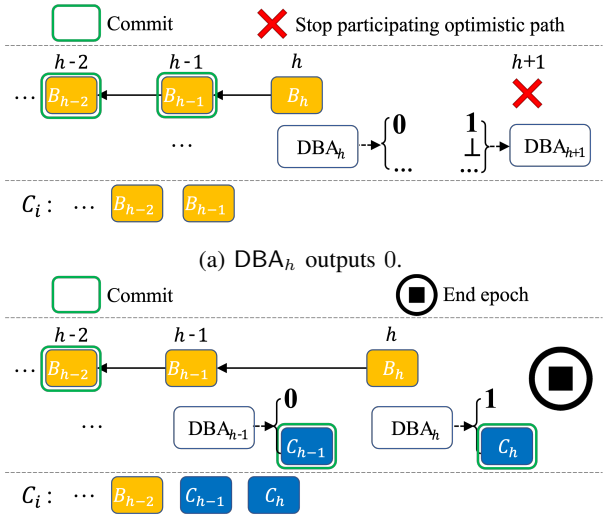
// optimistic path
3 **if** p_i *is* L_1 **then**
4 $\quad B_1 \leftarrow GenOptBlk(1, \perp)$; **broadcast** B_1

// pessimistic path
5 $InvokeDBA(1, 0, \perp)$

// optimistic path
6 **on** *receiving* B_1**:**
7 $\quad$ **send** $\mathsf{SignShr}_{n-t}(B_1)$ to L_2
8 **on** *receiving* n-t *sign. shares on* B_k *(denoted as* S*)***:**
9 $\quad$ **if** p_i *is* L_{k+1} **then**
10 $\quad\quad qc \leftarrow \mathsf{Comb}_{n-t}(B_k, S)$; $B_{k+1} \leftarrow GenOptBlk(k{+}1, qc)$;
11 $\quad\quad$ **broadcast** B_{k+1}
12 **while** *the epoch is not concluded***:**
13 $\quad$ **wait until** B_{h+1} is received or DBA_h outputs O
14 $\quad$ **if** B_{h+1} *is received before* DBA_h *outputs* **then**
$\quad\quad$ // optimistic path
15 $\quad\quad Commit(B_{h-1})$ **if** $h \geq 2$;
16 $\quad\quad$ **send** $\mathsf{SignShr}_{n-t}(B_{h+1})$ to L_{h+2};
17 $\quad\quad$ **stop** participating in DBA_{h-1} **if** $h \geq 2$;
18 $\quad\quad InvokeDBA(h+1, 0, B_{h+1}.QC)$;
19 $\quad\quad$ **broadcast** B_{h+1} if it has not broadcast yet
20 $\quad$ **else**
$\quad\quad$ // pessimistic path
21 $\quad\quad$ **if** $O.b = 0$ **then**
22 $\quad\quad\quad$ **stop** participating in the optimistic path;
23 $\quad\quad\quad InvokeDBA(h+1, 1, \perp)$;
24 $\quad\quad\quad$ **if** $h \geq 2$ *and* B_{h-1} *isn't committed* **then**
25 $\quad\quad\quad\quad$ **wait** to receive B_{h-1} and $Commit(B_{h-1})$;
26 $\quad\quad\quad prevPessBlk \leftarrow O.C$
27 $\quad\quad$ **else**
28 $\quad\quad\quad Commit(prevPessBlk)$; $Commit(O.C)$;
29 $\quad\quad\quad$ **conclude** the epoch
30 $\quad h \leftarrow h + 1$

2) Case 2: $t_1 \geq t_2$: In this case, DBA_h outputs before receiving B_{h+1}. When DBA_h outputs 0, it indicates an agreement that the optimistic path has been functioning well until height $h-1$. However, from this one replica's perspective, something is wrong with the optimistic path at height h. So the replica conveys this opinion by inputting 1 to the next DBA instance DBA_{h+1} and stops participating in the optimistic path. Conversely, if DBA_h outputs 1, signifying agreement among replicas that a failure has occurred with the optimistic path, the replica concludes the current epoch after committing pess-blocks. To delve into more details, we consider two sub-cases.

Case 2.1: DBA_h outputs 0. As illustrated in Figure 5a and detailed in Lines 21-22 of Algorithm 3, p_i promptly stops



(a) DBA_h outputs 0.

(b) DBA_h outputs 1.

Fig. 5: Actions taken when $t_1 \geq t_2$ in Ipotane

participating in the optimistic path of this epoch. Additionally, it inputs 1 to the subsequent DBA instance, expressing its opinion that the optimistic path has failed (Line 23). It can also commit the block B_{h-1} (when $h \geq 2$). If it has not received B_{h-1} yet, it will wait for the reception of B_{h-1} and then commit B_{h-1}. The pseudocode for this case is described in Lines 24-25. Furthermore, the pess-block C_h outputted from DBA_h, will be cached for now (Line 26) and will be committed later if the subsequent DBA instance outputs 1.

Case 2.2: DBA_h outputs 1. This sub-case indicates agreement among replicas that the optimistic path has failed. Consequently, every replica within this sub-case commits two pess-blocks and then concludes the current epoch. Actions taken by p_i are presented in Figure 5b and Lines 27-29 of Algorithm 3. After consecutively committing the opt-blocks until B_{h-2}, p_i commits two pess-blocks, C_{h-1} and C_h. Notably, C_{h-1} has been cached in the variable $prevPessBlk$, and C_h is outputted from DBA_h. Subsequently, p_i concludes its participation in the current epoch and progresses to the next epoch.

D. Detailed design when $h = 1$

In the initial opt-block B_1, QC for the preceding block is set to an empty value $\perp$ (Line 4 in Algorithm 3), following the approach in 2-chain HotStuff [14]. The first DBA instance DBA_1 is invoked with a binary input of 0 and QC set to the empty value $\perp$, as outlined in Line 5. Any replica that receives a message in the form of $(0, \perp, \mathsf{SignShr}_{t+1}(0))$ during the first round of the DBA_1 instance will straightforwardly recognize this binary input of 0 as valid.

V. Analysis of Ipotane

Our analysis of Ipotane covers two main aspects: correctness and efficiency. Correctness analysis examines whether Ipotane fulfills SMR's three properties, namely consistency, liveness, and completeness, which rely on some lemmas. Due

to space constraints, detailed proofs of these lemmas are provided in Appendix B.

A. Consistency analysis

To aid presentation, we denote an iteration of the loop (Lines 13-29 in Algorithm 3) with the parameter h as $iter_h$. Theorem 6 addresses the consistency property, supported by Lemmas 1, 2, 3, 4, and 5.

LEMMA 1. *If a non-faulty replica concludes an epoch in iteration $iter_h$, all non-faulty replicas will also conclude that epoch in $iter_h$.*

LEMMA 2. *Within an epoch, if a non-faulty replica commits an opt-block at height h and another non-faulty replica outputs b from DBA_{h+1}, then b must be 0.*

LEMMA 3. *Within an epoch, if two non-faulty replicas commit two blocks at the same height, then either both blocks are opt-blocks, or both are pess-blocks.*

LEMMA 4. *Within an epoch, if two non-faulty replicas commit two blocks at the same height, these two blocks must be identical.*

LEMMA 5. *If two non-faulty replicas conclude the same epoch, they must commit the same number of blocks within that epoch.*

THEOREM 6 (CONSISTENCY). *For two non-faulty replicas p_i and p_j, if $\mathcal{C}_i[k] \neq \perp$ and $\mathcal{C}_j[k] \neq \perp$, then $\mathcal{C}_i[k] = \mathcal{C}_j[k]$.*

Proof. Lemma 5 states that the epoch in which p_i commits $\mathcal{C}_i[k]$ must be the same as the epoch where p_j commits $\mathcal{C}_j[k]$. Furthermore, within that epoch, the height at which p_i commits $\mathcal{C}_i[k]$ must be the same as the height where p_j commits $\mathcal{C}_j[k]$. In other words, p_i and p_j commit $\mathcal{C}_i[k]$ and $\mathcal{C}_j[k]$ at the same height within the same epoch. According to Lemma 4, $\mathcal{C}_i[k]$ and $\mathcal{C}_j[k]$ must be identical. □

B. Liveness analysis

We say the protocol concludes an epoch if any non-faulty replica concludes it. By Lemma 1, non-faulty replicas agree on the number of iterations in each epoch. A transaction is considered committed if it is included in a committed block.

As described in Section II-B, each replica's buffer arranges pending transactions in the order of their reception times. Therefore, a unique index k, starting from 1, is assigned to each transaction within buf_i. Each time a block is committed, any transaction included in this block will be removed from buf_i, and the indices of remaining transactions are adjusted downwards. Recall that in Section II-B, the maximum transaction count in a block is denoted as c. For a given transaction tx in replica p_i's buffer, the committing of p_i's newly proposed block results in one of two outcomes for tx: either tx is included within the block and becomes committed, or the index of tx decreases by c. To unify the two cases, we define tx as *committed* when tx's index becomes 0 or negative.

Consider the moment when tx enters the buffer of every non-faulty replica and suppose tx is placed at index k_i in replica p_i's buffer. Whenever an index k_i falls to 0 or below, tx is committed by p_i. Let K represent the sum of tx's indices in the buffers of all non-faulty replicas, expressed as $K = \sum_{p_i \in H} k_i$, where H is the set of non-faulty replicas. It follows naturally that each time a block from a non-faulty replica is committed, K decreases.

The liveness property is outlined in Theorem 8, whose proof relies on Lemma 7.

LEMMA 7. *If a non-faulty replica commits a block, every non-faulty replica will eventually commit this block.*

THEOREM 8 (LIVENESS). *If a transaction tx is added to every non-faulty replica's buffer, every non-faulty replica will eventually commit a block containing tx.*

Proof. Let T_0 denote the moment when tx is added to every non-faulty replica's buffer. Let $u = \lceil K/c \rceil$. Two situations unfold:

Situation 1: At least u non-faulty opt-blocks proposed after T_0 are committed within an epoch. Each time a non-faulty opt-block is committed, K will be reduced by c. Therefore, after u non-faulty opt-blocks are committed, K will be reduced by $c \cdot u$. Since $u = \lceil K/c \rceil$, $K - u \cdot c \leq 0$. As K represents the sum of all indices of tx in non-faulty replicas' buffers, at least one index is negative or 0, indicating that tx is committed by some non-faulty replica. Denote this non-faulty replica as p_i, which commits a block B containing tx. According to Lemma 7, each non-faulty replica will also commit B.

Situation 2: Less than u non-faulty opt-blocks proposed after T_0 are committed within each epoch. In this situation, each epoch is concluded after some opt-blocks and two pess-blocks are committed. DBA's quality property ensures that the probability of the outputted pess-block being proposed by a non-faulty replica is over $1/2$. Similar to Situation 1, each time a non-faulty pess-block is committed, K will be reduced by c. As the epochs advance, the probability that at least u non-faulty pess-blocks are committed will approach 1. In other words, K will keep decreasing and eventually become negative or 0. Thus, tx will eventually be committed by some non-faulty replica. By Lemma 7, every non-faulty replica will also commit tx. □

C. Completeness analysis

THEOREM 9 (COMPLETENESS). *For two non-faulty replicas p_i and p_j, if $\mathcal{C}_i[k] \neq \perp$, then for each index $k' \leq k$, eventually $\mathcal{C}_j[k'] \neq \perp$.*

Proof. According to the sequential rule outlined in Section II-B, for each index k' ($k' \leq k$), it must hold that $\mathcal{C}_i[k'] \neq \perp$. If $\mathcal{C}_i[k']$ is committed through the optimistic path, based on Line 19 of Algorithm 3, p_j will eventually commit a block identical to $\mathcal{C}_i[k']$. Conversely, if $\mathcal{C}_i[k']$ is committed via the pessimistic path, the termination property of DBA ensures that p_j will also eventually commit a block identical to $\mathcal{C}_i[k']$.

To sum up, p_j will eventually commit k blocks. In other words, for each index k' ($k' \leq k$), $\mathcal{C}_j[k'] \neq \perp$ will hold eventually. □

D. Efficiency analysis

Recall that δ denotes the actual network delay, while c and L represent the maximum transaction count and block size of a block, respectively. Additionally, we assume the size of shares and signatures to all have length κ. Our analysis focuses on the efficiency of Ipotane when employing sMVBA [24] to construct AlgDBA. Inspired by AMS-VABA [28] and 2-chain VABA [14], we introduce two improvements to sMVBA. Firstly, we reduce its view-change phase from two communication rounds to just one, in a manner akin to AMS-VABA [28], effectively reducing its expected worst-case latency to 10.5 rounds. Secondly, we require each replica to broadcast a block within the second *Provable Broadcast* (PB) instance. For clarity, we refer to these blocks as PB2-blocks. Accordingly, original pess-blocks proposed in the first PB instance are termed PB1-blocks. When a replica commits the PB1-block proposed by the view leader (distinct from the leader of Ipotane's optimistic path), it must have received a QC for this leader's PB2-block. The replica will include this QC in its PB1-block in the subsequent sMVBA/AlgDBA instance, leading to a chain of blocks across sMVBA/AlgDBA instances, similar to 2-chain VABA [14]. This way, committing a PB1-block in an AlgDBA will also commit a PB2-block from the preceding AlgDBA, thus improving DBA's throughput.

Before analyzing the efficiency of the protocol, we first provide a rigorous definition of favorable/unfavorable situations.

1) Definitions of favorable/unfavorable situations: A favorable situation is defined by two conditions:

- The leader on the optimistic path is non-faulty, and
- the message delays on the optimistic path are bounded relative to the pessimistic path. Specifically, if we denote the delay of pessimistic-path messages as δ_p, then optimistic-path messages have delays of at most $3.5\delta_p$.

The second condition ensures that optimistic-path messages are not substantially slower than their pessimistic-path counterparts, thereby preventing the pessimistic path from overtaking the optimistic path.

Conversely, a situation is considered unfavorable if either:

- The optimistic-path leader is faulty, or
- the optimistic-path message delays exceed $3.5\delta_p$.

2) Analysis of favorable situations: An opt-block gets certified in just two communication rounds, while a pess-block requires seven rounds. In a favorable situation, since optimistic-path messages have delays at most 3.5 times larger than those on the pessimistic path, opt-blocks always get certified earlier than their corresponding pess-blocks. As a result, blocks are continuously committed through the optimistic path.

Every 2δ interval, a new opt-block is produced, and a block from two heights prior is committed. This process results in a throughput of $c/(2\delta)$ and a latency of 5δ. Even in this favorable situation, both paths are executed. On the optimistic path, each replica will send signature shares to leaders and broadcast its received opt-block, leading to a communication overhead of $O(n^2L + n\kappa)$. The pessimistic path consists of consecutive AlgDBA instances, leading to an overhead of $O(n^2L + n^2\kappa)$. Therefore, the total communication overhead for Ipotane in a favorable situation is $O(n^2L + n^2\kappa)$. While protocols such as 2-chain HotStuff or Ditto have a lower, linear communication overhead of $O(nL+n\kappa)$ under favorable situations, our experiments in Section VI demonstrate that this linear communication does not materialize as practical efficiency gains.

3) Analysis of unfavorable situations: In an unfavorable scenario, pess-blocks are certified earlier than their corresponding opt-blocks. Each non-faulty replica will input 1 to the second AlgDBA instance, subsequently producing an output of 1 from it. Therefore, the replica commits blocks outputted from the two AlgDBA instances. For clarity, we refer to these instances as AlgDBA_1 and AlgDBA_2, respectively. As AlgDBA is constructed as an extension of sMVBA with an additional communication round, its expected worst-case latency is 11.5 rounds. At the end of an epoch, three blocks are committed: two PB1-blocks generated in AlgDBA_1 and AlgDBA_2, respectively, and one PB2-block generated in AlgDBA_1. Consequently, the throughput is calculated as $3c/(11.5\delta \cdot 2) = 3c/(23\delta)$. The latency for the first PB1-block is 23δ, corresponding to the duration of two AlgDBA instances. The PB2-block, proposed two rounds later than the first PB1-block, has a latency of 21δ. The second PB1-block, committed immediately upon the output of AlgDBA_2, has a latency of 11.5δ. Therefore, the average latency across these blocks is $(23\delta + 21\delta + 11.5\delta)/3$, which equals 18.5δ. As for the communication overhead, the optimistic path in the unfavorable situation fails to make progress. Therefore, its communication overhead is that of the pessimistic path, which is also $O(n^2L + n^2\kappa)$.

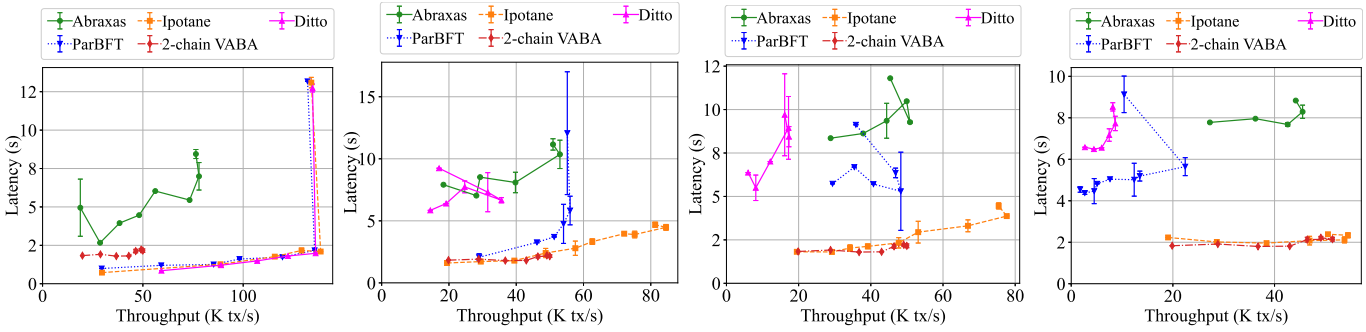
VI. Implementation and Evaluation

In this section, we present the implementation of Ipotane and conduct a comparison with other protocols. Our chosen baselines include Abraxas and ParBFT, both of which employ the parallel-path paradigm similar to Ipotane. We include Ditto as another baseline that represents the sequential-path paradigm. In favorable situations, Ditto's performance matches that of a partially-synchronous protocol. We also include 2-chain VABA, a purely asynchronous protocol, as a baseline for the evaluation of unfavorable situations.

A. Implementation and experimental setup

1) Implementation: We directly adopt the available open-source codes of our baselines ParBFT[4] and Abraxas[5]. 2-chain VABA and Ditto share the same repository[6]. All these implementations are built on the same code framework in Rust, which typically includes a mempool to decouple transaction transmission from consensus messages. Through mempool, each replica continuously packages a batch of transactions into a payload, which is then broadcast to others. In the consensus message, a block contains only hashes of these

[4]https://github.com/ac-dcz/parbft-parbft1-rust
[5]https://github.com/sochsenreither/abraxas
[6]https://github.com/danielxiangzl/Ditto



(a) $\rho = 0$ (b) $\rho = 10\%$ (c) $\rho = 20\%$ (d) $\rho = 100\%$

Fig. 6: Latency vs. throughput

payloads, effectively reducing the size of consensus messages and enhancing performance.

To ensure a fair comparison, we implement Ipotane using the same framework as the baselines, and make our implementation publicly available[7]. The VABA protocol employed in DBA is instantiated with sMVBA [24]. To improve performance, we introduce minor modifications to sMVBA. These include adding a block in each PB instance and chaining blocks across different sMVBA instances, a structure adopted by 2-chain VABA [14]. In addition, the view-change phase has been streamlined to a single round following AMS-VABA [28].

2) Experimental setup: For all protocols, we set the size of a transaction to 512 bytes. The size of a payload and the queue capacity in the mempool are configured to be 500 kilobytes and 100,000, respectively. The maximum number of payloads contained in a block is limited to 32, and the minimum interval to propose a payload is set to 100 milliseconds. Following the configuration from the Ditto paper [14], we set its timing parameter Δ to 5 seconds. Similarly, we adopt the settings from the Abraxas paper [17], configuring its lookback parameter λ to 20.

Except for the 2-chain VABA, each protocol employs predetermined leaders for optimistic paths. Depending on the leader crash frequency, we consider the following three scenarios, akin to those defined in Abraxas. Each scenario is characterized by the parameter ρ, signifying the probability of leader crashes.

1) $\rho = 0$: This implies that leaders operate without crashes. In this scenario, all protocols, except 2-chain VABA, are expected to commit blocks through optimistic paths.
2) $\rho = 100\%$: In this scenario, leaders always crash, and all protocols commit blocks through pessimistic paths.
3) $\rho = 10\%$ or $\rho = 20\%$: Each leader has a 10% or 20% probability of crashing in this scenario, representing an intermediate point between the previous two scenarios. Optimistic protocols commit blocks through their optimistic paths intermittently.

[7]https://github.com/CGCL-codes/ipotane

Our experiments are conducted on AWS, where each replica is deployed as an m5d.2xlarge instance. Each instance is equipped with 8 vCPUs and 32GB of memory, running Ubuntu 20.04. Replicas are connected through a network link with up to 10 Gbps bandwidth. These replicas are spread in a geographically distributed manner, uniformly across five regions: N. Virginia, Stockholm, Tokyo, Sydney, and N. California.

3) Performance metrics: Our evaluation focuses on two key metrics: end-to-end latency and throughput. End-to-end latency is assessed as the average time taken for a transaction to be committed, measured from the moment it is submitted by the client to the moment it is committed. Throughput is calculated as the number of committed transactions per second. Each experiment is conducted over a duration of 5 minutes to report a stable performance. We repeat each experiment three times and utilize error bars or averages to mitigate experimental errors.

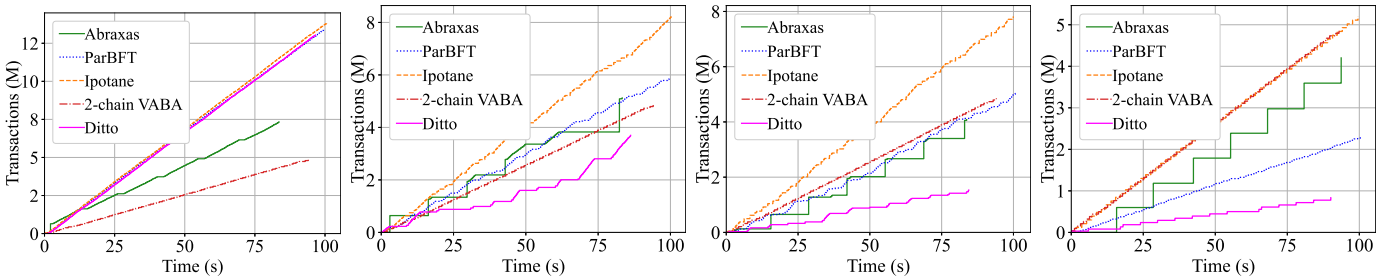
B. Trade-off between throughput and latency

In all experiments in this section, we set the number of replicas to 16. By progressively increasing the rate at which clients submit transactions, the system eventually becomes saturated. Plotting each pair of latency and throughput produces a figure that simultaneously demonstrates the latency under unsaturated conditions and the peak throughput under saturated conditions. Experimental results are illustrated in Figure 6, with throughput and latency on the x-axis and y-axis, respectively. Each data point in the figure is marked with an error bar, representing both the average and standard deviation of the experimental results.

As shown in Figure 6a, when the optimistic path always operates well, Ipotane attains low latency and high throughput, comparable to Ditto and ParBFT.[8] Notably, Ditto matches a partially-synchronous protocol's performance, as it adopts the sequential-path paradigm and only runs the optimistic path in this scenario. Thus, in favorable situations, Ipotane's performance is on par with a partially-synchronous protocol.

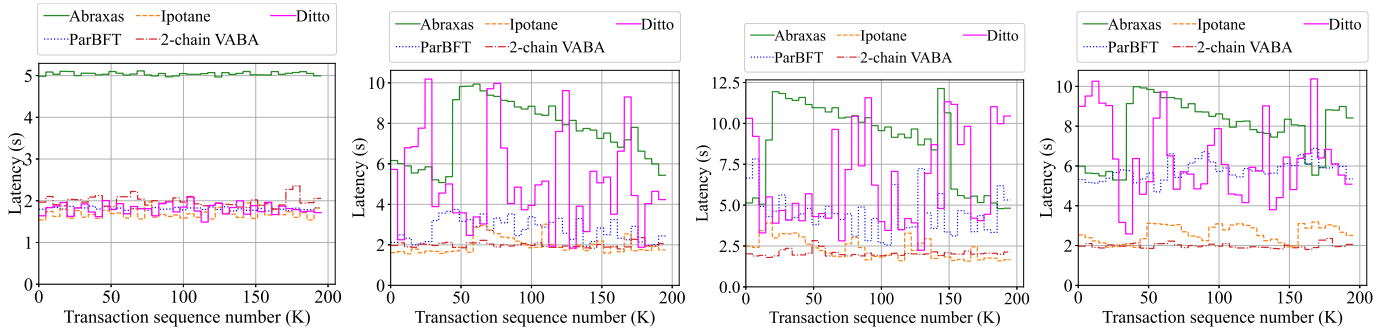
At the other end of the spectrum, when the optimistic path always fails, as shown in Figure 6d, Ipotane still maintains

[8]Abraxas reports lower performance than expected, possibly due to its implementation being based on an earlier version of Ditto.



(a) $\rho = 0$ (b) $\rho = 10\%$ (c) $\rho = 20\%$ (d) $\rho = 100\%$

Fig. 7: Throughput over time



(a) $\rho = 0$ (b) $\rho = 10\%$ (c) $\rho = 20\%$ (d) $\rho = 100\%$

Fig. 8: Latency over the transaction sequence numbers

good performance, slightly inferior to the purely asynchronous protocol (2-chain VABA) but significantly better than Ditto or ParBFT. In this scenario, Ditto takes a considerable amount of time to switch between the failed optimistic path and the pessimistic path, resulting in poor performance. While ParBFT runs two paths concurrently, it requires additional ABA instances to commit pess-blocks. These ABA instances do not generate new blocks by themselves, leading to idle periods and reduced performance. In contrast, Ipotane commits pess-blocks by consecutively running DBA instances, which can promptly detect the optimistic path's failure without introducing extra consensus instances, thereby delivering superior performance.

In the intermediate scenarios, a protocol with an optimistic path intermittently commits blocks through this path, leading to a blended result between the scenarios of $\rho = 0$ and $\rho = 100\%$. Regarding peak throughput, Ipotane consistently outperforms others as illustrated in Figure 6b and Figure 6c. In terms of latency, Ipotane and 2-chain VABA consistently demonstrate the lowest among these protocols.

To sum up, across all scenarios, Ipotane consistently attains the (near-)best performance among all protocols. Specifically, it achieves performance on par with partially-synchronous protocols in favorable situations and on par with purely asynchronous ones in unfavorable situations.

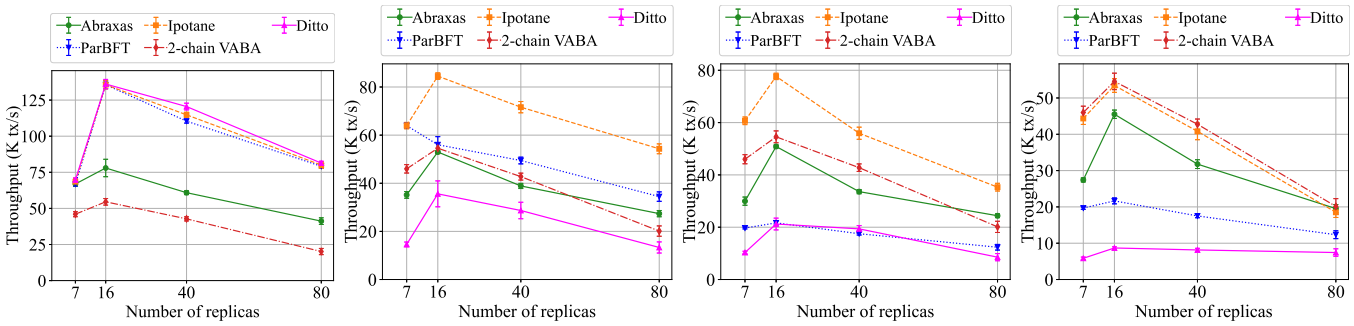
C. Throughput stability

In this section, we continue to use 16 replicas. Our experiments are specifically conducted at each system's saturation point, where a system achieves its peak throughput without significant deterioration in latency. At this point, the system can reliably sustain high throughput.

Starting from the moment the system reaches the saturation point, we record the accumulated number of committed transactions over time. More precisely, each time a new block is committed, we record the current time and calculate the number of committed transactions by counting in transactions included in this block. We also explore four scenarios with varying values of ρ, whose results are depicted in Figure 7.

In the $\rho = 0$ scenario, all protocols exhibit stable throughput, as evidenced by the smooth curves in Figure 7a. This is expected, as the 2-chain VABA continuously commits blocks through the two-chain instances, while other protocols steadily commit blocks through the optimistic path. On the other hand, in the $\rho = 100\%$ scenario, 2-chain VABA, ParBFT, and Ipotane can maintain stable throughput, as shown in Figure 7d. However, Ditto and Abraxas display unstable throughput, as indicated by the jagged curves. This instability arises from the extended periods required for Ditto to complete the path switch and for Abraxas to wait for a minimum of λ pess-blocks, during which no blocks are being committed.

In the $\rho = 10\%$ or $\rho = 20\%$ scenario, all protocols except VABA exhibit less stable throughput as they alternate between committing blocks through the optimistic path and the pessimistic path. Nevertheless, Ipotane continues to showcase superior stability than Abraxas and Ditto.



(a) $\rho = 0$ (b) $\rho = 10\%$ (c) $\rho = 20\%$ (d) $\rho = 100\%$

Fig. 9: Throughput vs. system size

D. Latency stability

Latency stability holds significant importance for upper-layer applications, as unstable latency can result in poor user experience. We evaluate latency stability by recording each transaction's latency. These experiments are also conducted with 16 replicas and at each system's saturation point.

Experimental results are depicted in Figure 8. In the $\rho = 0$ scenario (Figure 8a), all protocols exhibit stable latency. In the $\rho = 100\%$ scenario (Figure 8d), Ipotane maintains relatively stable latency by committing pess-blocks through successive DBA instances. While Ipotane's latency deviation is slightly larger than that of 2-chain VABA, it is more stable than the others. Ipotane's slightly larger deviation than VABA can be attributed to the fact that, within an epoch, pess-blocks generated in the second-to-last DBA instance are not committed until the final DBA instance outputs, resulting in higher latency for these pess-blocks. In contrast, Abraxas displays significant latency fluctuations due to its lookback mechanism, where blocks generated in the initial two-chain instance must wait for the production of λ subsequent blocks. ParBFT and Ditto both exhibit notable latency instability, because their respective ABA instances and path-switch mechanisms introduce large latency variations.

In the $\rho = 10\%$ (Figure 8b) and $\rho = 20\%$ (Figure 8c) scenarios, Ipotane and VABA still maintain stable latency, with fluctuation significantly lower than other protocols.

E. Scalability evaluation

We comprehensively evaluate scalability across various protocols, analyzing their throughput under varying numbers of replicas: 7 replicas, 16 replicas, 40 replicas, and 80 replicas. Throughput measurements are specifically taken at the saturation point. Additionally, experiments are conducted with different probabilities of leader replicas, whose results are shown in Figure 9.

As depicted in Figure 9a, Ipotane, alongside ParBFT, achieves a high throughput when leaders on the optimistic path keep performing well. Notably, in the case of 40 or 80 replicas, they exhibit a slightly lower throughput compared to Ditto, potentially attributed to their elevated communication overhead $O(n^2)$, stemming from the parallel pessimistic path. In scenarios where the optimistic path fails to function, as illustrated in Figure 9d, Ipotane consistently maintains high throughput comparable to 2-chain VABA, across different replica counts. When leaders on the optimistic path fail with a probability of 10% or 20%, Ipotane outperforms all other protocols under varying system sizes, as evidenced by Figure 9b or Figure 9c. In summary, Ipotane consistently demonstrates excellent scalability across diverse probabilities of leader failures.

VII. Related Work

We summarize asynchronous BFT protocols in this section and defer the discussion of (partially-)synchronous protocols to Appendix C.

The simplest form of asynchronous BFT is ABA, which reaches agreements on binary values [29], [30], [31], [32]. VABA and MVBA instead focus on agreeing on arbitrary values [21], [22], [23], [33]. Building upon ABA or VABA, *Asynchronous Common Subset* (ACS) and SMR can be constructed [10], [34], [35], [36].

Despite efforts to enhance the performance of asynchronous protocols, a performance gap persists when compared to partially-synchronous protocols. To address this gap, a series of works introduce an optimistic path to asynchronous protocols, categorized into two paradigms: sequential-path and parallel-path. The sequential-path paradigm executes the optimistic and pessimistic paths in sequence [37], [38], [39], necessitating path switches [14], [15]. These switches delay the launch of the pessimistic path and affect performance in unfavorable situations. To overcome this, the parallel-path paradigm, exemplified by Abraxas [17] and ParBFT [16], launches two paths simultaneously, avoiding the need for path switches. However, while Abraxas achieves high throughput in all situations, it suffers from high latency under unfavorable situations. In contrast, ParBFT consistently delivers low latency but suffers from reduced throughput in unfavorable situations. Ipotane proposed in this paper achieves both high throughput and low latency in both favorable and unfavorable situations.

Another class of protocols [40], [41], [42] leverages a *Directed Acyclic Graph* (DAG)-based approach. These protocols,

however, inherently suffer from $O(n^2L+n^3\kappa)$ communication overhead, rendering them less scalable than many previously discussed protocols. For instance, Ipotane requires a communication overhead of only $O(n^2L + n^2\kappa)$. Moreover, these approaches generally depend on multiple rounds of *Reliable Broadcast* (RBC) to commit, resulting in high latency. For instance, DAGRider and Tusk require latencies of 12δ and 9δ, respectively, even under favorable situations. BullShark [43], a noteworthy DAG-based protocol, also introduces an optimistic path to enhance performance. In favorable situations, it requires two sequential RBCs to commit, incurring a latency of 6δ, which is slightly larger than 5δ offered by a partially-synchronous protocol (e.g., 2-chain HotStuff) or our Ipotane. However, it has a complex process of transitioning to the pessimistic path in unfavorable situations, resulting in an expected latency of 30δ due to 10 sequential RBCs. This is significantly higher than 10.5δ typical of purely asynchronous protocols (e.g., 2-chain VABA) or 18.5δ offered by Ipotane.

VIII. Conclusion

Existing dual-path asynchronous BFT protocols exhibit either low throughput or high latency under unfavorable situations. To address this, we propose a novel protocol named Ipotane, which executes consecutive DBA instances on the pessimistic path. DBA operates as a fusion of biased ABA and VABA, which can be implemented through low-cost modifications to existing VABA protocols. On one hand, DBA promptly detects optimistic path failures, ensuring low latency under unfavorable situations. On the other hand, Ipotane leverages DBA instances to continuously produce blocks without idle periods, thereby achieving high throughput in unfavorable situations. In summary, Ipotane attains performance on par with partially-synchronous protocols under favorable situations and comparable to purely asynchronous protocols in unfavorable situations, as demonstrated by our experiments.

Acknowledgment

This work is supported by National Science and Technology Major Project 2022ZD0115301. This work is also funded by the European Union, ERC-2023-STG, Project ID: 101116713. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 183–187, 2017.

[2] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[3] Z. Wu, J. Liu, J. Wu, Z. Zheng, X. Luo, and T. Chen, "Know your transactions: Real-time and generic transaction semantic representation on blockchain & web3 ecosystem," in *Proceedings of the ACM Web Conference*. ACM, 2023, pp. 1918–1927.

[4] J. J. Si, T. Sharma, and K. Y. Wang, "Understanding user-perceived security risks and mitigation strategies in the web3 ecosystem," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 2024, pp. 1–22.

[5] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020.

[6] X. Wang, S. Duan, J. Clavin, and H. Zhang, "BFT in blockchains: From protocols to use cases," *ACM Computing Surveys*, vol. 54, no. 10, pp. 1–37, 2022.

[7] S. Duan, M. K. Reiter, and H. Zhang, "BEAT: Asynchronous BFT made practical," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2028–2041.

[8] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "DispersedLedger: High-throughput Byzantine consensus on variable bandwidth networks," in *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2022, pp. 493–512.

[9] D. S. Antunes, A. N. Oliveira, A. Breda, M. G. Franco, H. Moniz, and R. Rodrigues, "Alea-BFT: Practical asynchronous Byzantine fault tolerance," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2024, pp. 313–328.

[10] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 31–42.

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

[12] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 1999, pp. 173–186.

[13] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 347–356.

[14] R. Gelashvili, L. Kokoris Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Proceedings of the 26th International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 296–315.

[15] Y. Lu, Z. Lu, and Q. Tang, "Bolt-Dumbo transformer: Asynchronous consensus as fast as the pipelined BFT," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 2159–2173.

[16] X. Dai, B. Zhang, H. Jin, and L. Ren, "ParBFT: Faster asynchronous BFT consensus with a parallel optimistic path," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 504–518.

[17] E. Blum, J. Katz, J. Loss, K. Nayak, and S. Ochsenreither, "Abraxas: Throughput-efficient hybrid asynchronous consensus," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 519–533.

[18] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, "Fast-HotStuff: A fast and robust BFT protocol for blockchains," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 2478–2493, 2023.

[19] B. Libert, M. Joye, and M. Yung, "Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares," in *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*. ACM, 2014, pp. 303–312.

[20] R. Bacho and J. Loss, "On the adaptive security of the threshold BLS signature scheme," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 193–207.

[21] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Proceedings of the 21st Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[22] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous Byzantine agreement with expected O(1) rounds, expected communication, and optimal resilience," in *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 320–334.

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

[24] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding Dumbo: Pushing asynchronous BFT closer to practice," *Cryptology ePrint Archive*, 2022.

[25] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1983, pp. 27–30.
[26] R. Friedman, A. Mostefaoui, and M. Raynal, "Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 46–56, 2005.
[27] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
[28] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous Byzantine agreement," in *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 337–346.
[29] M. O. Rabin, "Randomized Byzantine generals," in *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*. IEEE, 1983, pp. 403–409.
[30] S. Toueg, "Randomized Byzantine agreements," in *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1984, pp. 163–178.
[31] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages," in *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*. ACM, 2014, pp. 2–9.
[32] I. Abraham, N. Ben-David, and S. Yandamuri, "Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement," in *Proceedings of the 41st ACM Symposium on Principles of Distributed Computing*. ACM, 2022, pp. 381–391.
[33] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-NG: Fast asynchronous BFT consensus with throughput-oblivious latency," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 1187–1201.
[34] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous BFT protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020, pp. 803–818.
[35] H. Zhang and S. Duan, "PACE: Fully parallelizable BFT from reproposable Byzantine agreement," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 3151–3164.
[36] S. Duan, X. Wang, and H. Zhang, "Fin: Practical signature-free asynchronous common subset in constant time," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 815–829.
[37] K. Kursawe and V. Shoup, "Optimistic asynchronous atomic broadcast," in *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*. Springer, 2005, pp. 204–215.
[38] H. V. Ramasamy and C. Cachin, "Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast," in *Proceedings of the 24th International Conference On Principles Of Distributed Systems*. Springer, 2005, pp. 88–102.
[39] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *Proceedings of the 5th European Conference on Computer Systems*. ACM, 2010, pp. 363–376.
[40] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*. ACM, 2021, pp. 165–175.
[41] M. A. Schett and G. Danezis, "Embedding a deterministic BFT protocol in a block DAG," in *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*. ACM, 2021, pp. 177–186.
[42] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus," in *Proceedings of the 17th European Conference on Computer Systems*. ACM, 2022, pp. 34–50.
[43] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2022, pp. 2705–2718.
[44] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, 1980.
[45] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.
[46] D. Dolev, M. J. Fischer, R. Fowler, N. A. Lynch, and H. R. Strong, "An efficient algorithm for Byzantine agreement without authentication," *Information and Control*, vol. 52, no. 3, pp. 257–274, 1982.
[47] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series, consensus system," *arXiv preprint arXiv:1805.04548*, 2018.
[48] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync HotStuff: Simple and practical synchronous state machine replication," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 106–118.
[49] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.
[50] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. ACM, 2007, pp. 45–58.
[51] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Scalable Byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 139–151, 2018.
[52] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. A. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: A scalable and decentralized trust infrastructure," in *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2019, pp. 568–580.
[53] X. Dai, L. Huang, J. Xiao, Z. Zhang, X. Xie, and H. Jin, "Trebiz: Byzantine fault tolerance with Byzantine merchants," in *Proceedings of the 38th Annual Computer Security Applications Conference*. ACM, 2022, pp. 923–935.
[54] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.
[55] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.
[56] J. Niu, F. Gai, M. M. Jalalzai, and C. Feng, "On the performance of pipelined HotStuff," in *Proceedings of the 40th Annual IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.
[57] N. Giridharan, F. Suri-Payer, M. Ding, H. Howard, I. Abraham, and N. Crooks, "Beegees: stayin'alive in chained BFT," in *Proceedings of the 42nd ACM Symposium on Principles of Distributed Computing*. ACM, 2023, pp. 233–243.
[58] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks, "Motorway: Seamless high speed BFT," *arXiv preprint arXiv:2401.10369*, 2024.
[59] M. J. Amiri, C. Wu, D. Agrawal, A. El Abbadi, B. T. Loo, and M. Sadoghi, "The bedrock of Byzantine fault tolerance: A unified platform for BFT protocols analysis, implementation, and experimentation," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2024, pp. 371–400.

APPENDIX A
CORRECTNESS ANALYSIS OF AlgDBA

In this section, we prove our AlgDBA construction adheres to all the properties outlined in Section III-A.

1) Agreement, quality, and block-external validity: These properties of AlgDBA are derived directly from VABA.

2) Termination: AlgDBA's termination property is stated in Theorem 11, supported by Lemma 10.

LEMMA 10. *Every non-faulty replica will receive either $t+1$ values of 0 or $n-t$ values of 1 during the first communication round in* AlgDBA.

Proof. This is established through two cases.

Case 1: At least one non-faulty replica has the binary input of 0. In this case, each non-faulty replica will receive a message containing 0. According to Lines 6-8 in Algorithm 1, each non-faulty replica will also broadcast a message with 0

if it has not yet broadcast this message. Therefore, each non-faulty replica will eventually receive $t+1$ values of 0 during the first round.

Case 2: Every non-faulty replica has a binary input of 1. Here, each non-faulty replica broadcasts a message containing 1, leading to each receiving $n-t$ values of 1. □

THEOREM 11. AlgDBA *achieves termination.*

Proof. Based on Lemma 10, every non-faulty replica can generate a valid input for VABA. Following the termination property of VABA, all non-faulty replicas will eventually produce an output from VABA and thus from AlgDBA. □

3) Binary-external validity: Suppose by contradiction a non-faulty replica outputs $\langle 0, * \rangle$, but no replica inputs 0 with a valid proof σ. In such a case, no one can receive $t+1$ messages containing 0 to create a valid signature sig_0 or form a valid input of 0 to VABA, as described in Lines 9-12 in Algorithm 1. Thus, the binary output from VABA or AlgDBA cannot be 0, contradicting the initial assumption.

4) Biased validity: If at least $t+1$ non-faulty replicas input $\langle 0, * \rangle$, it implies that at most $n-t-1$ replicas, whether non-faulty or Byzantine, will input $\langle 1, * \rangle$. Hence, the condition in Line 13 of Algorithm 1 will not be met. Even a Byzantine replica cannot forge a valid threshold signature on 1. Therefore, every replica, whether non-faulty or Byzantine, can only input a tuple containing the bit 0 to VABA. This ensures that the output from VABA and AlgDBA will contain the bit 0, thus guaranteeing biased validity.

APPENDIX B
PROOF OF LEMMAS

LEMMA 1. *If a non-faulty replica concludes an epoch in iteration $iter_h$, all non-faulty replicas will also conclude that epoch in $iter_h$.*

Proof. If a non-faulty replica concludes an epoch in $iter_h$, it implies that DBA_h outputs 1. According to DBA's biased-validity property, at least $n-2t$ non-faulty replicas must have inputted 1 to DBA_h. Consequently, as per the rules of Case 2.1, these $n-2t$ non-faulty replicas must have stopped voting for the opt-block B_h. This means no valid QC_h can be generated, and no valid B_{h+1} can be constructed, effectively stopping on the optimistic path. On the other hand, a replica will input to DBA_{h+1} only after DBA_h outputs. Based on DBA's agreement property, every non-faulty replica will output 1 from DBA_h and consequently conclude the epoch in $iter_h$. □

LEMMA 2. *Within an epoch, if a non-faulty replica commits an opt-block at height h and another non-faulty replica outputs b from DBA_{h+1}, then b must be 0.*

Proof. We assume these two non-faulty replicas to be p_i and p_j where p_i commits an opt-block B_h and p_j receives b from DBA_{h+1}. B_h must be committed through the rules of either Case 1 or Case 2.1 in Section IV-C. If it is Case 1, at least $n-t$ replicas, among which $n-2t$ are non-faulty, must have voted for B_{h+1}. This implies that at least $n-2t$ non-faulty replicas would use 0 as the binary input to DBA_{h+1}. Since $n-2t \geq t+1$, the biased validity ensures DBA_{h+1} outputs a binary value of 0. If B_h is committed through Case 2.1, based on DBA's agreement property, p_j would also receive a binary output of 0 from DBA_{h+1}. □

LEMMA 3. *Within an epoch, if two non-faulty replicas commit two blocks at the same height, then either both blocks are opt-blocks, or both are pess-blocks.*

Proof. We prove this lemma via contradiction. Without loss of generality, assume two non-faulty replicas p_i and p_j commit opt-block B_h and pess-block C_h, respectively. Based on Lemma 2, p_j will receive 0 from DBA_{h+1} if it receives an output at all. According to the protocol described in Section IV-C, p_j must commit a pess-block C_{h-1} or C_{h+1}. We consider the following two situations:

Situation 1: p_j commits C_h and C_{h+1}. Per the rules of Case 2.2, p_j must receive a binary output of 1 from DBA_{h+1}, contradicting the earlier conclusion that DBA_{h+1} outputs 0.

Situation 2: p_j commits C_{h-1} and C_h. Per the rules of Case 2.2, p_j must receive a binary output of 1 from DBA_h. With the biased-validity property, at least $n-2t$ non-faulty replicas must have inputted 1 to DBA_h. Thus, per the rules of Case 2.1, these replicas would stop voting for the opt-block B_h, preventing the generation of a valid QC_h or B_{h+1}. This makes it impossible for p_i to commit B_h through Case 1. Additionally, p_j will conclude the current epoch in iteration $iter_h$. By Lemma 1, all non-faulty replicas will conclude the epoch in $iter_h$ without inputting to DBA_{h+1}, making it impossible for p_i to commit B_h through Case 2.1. This contradicts the assumption that p_i commits an opt-block B_h.

Therefore, it is impossible for one non-faulty replica to commit an opt-block and the other to commit a pess-block at the same height, establishing the lemma. □

LEMMA 4. *Within an epoch, if two non-faulty replicas commit two blocks at the same height, these two blocks must be identical.*

Proof. Per Lemma 3, either both blocks are opt-blocks, or both are pess-blocks. If they are pess-blocks, then according to DBA's agreement property, these two blocks must be identical. Now, we consider the situation where both are opt-blocks.

As described in Section IV-C, an opt-block B_h is committed through either Case 1 or Case 2.1. If B_h is committed through Case 1, a QC for B_h must be generated. If B_h is committed through Case 2.1, DBA_{h+1} must produce an output O_{h+1} where $O_{h+1}.b = 0$ and $O_{h+1}.d = B_h$. By DBA's binary-external validity property, a replica must have input $\langle 0, \sigma, * \rangle$ to DBA_{h+1}, where σ is the QC for B_h. In short, if an opt-block is committed, it must be certified by a QC.

Let the two committed opt-blocks be B_h and B'_h, certified by QC_h and QC'_h, respectively. By a standard quorum intersection argument on QC and QC', we have $B_h = B'_h$. □

LEMMA 5. *If two non-faulty replicas conclude the same epoch, they must commit the same number of blocks within that epoch.*

Proof. Based on Lemma 1, these two replicas must conclude the epoch in the same iteration, which we denote as $iter_l$. According to Algorithm 3, both replicas must receive 1 from DBA_l. Since every non-faulty replica inputs 0 to DBA_1 (as stated in Line 5 of Algorithm 3), the biased-validity property ensures that DBA_1 will output 0. Consequently, l must be equal to or greater than 2. Additionally, both replicas must have received 0 from the preceding DBA instance DBA_{l-1}.

At any height k, where $1 \leq k \leq l$-2, both replicas commit an opt-block B_k. At heights l-1 and l, they commit a pess-block, C_{l-1} or C_l, respectively. Therefore, these two replicas commit the same number of blocks in that epoch. □

LEMMA 7. *If a non-faulty replica commits a block, every non-faulty replica will eventually commit this block.*

Proof. Without loss of generality, assume that a non-faulty replica p_i commits a block B. If B is a pess-block, it must be committed through an output of 0 from DBA. By the termination property of DBA, each non-faulty replica will output 0 and commit B.

If B is an opt-block, assume p_i commits B at height h during an epoch. According to Line 19 in Algorithm 3, p_i will broadcast B, and each non-faulty replica will eventually receive B. For another non-faulty replica p_j, we consider the following two cases. If p_j receives B_{h+2} before C_{h+1}, then p_j will also commit B through the two-chain rule. Otherwise, namely if p_j receives C_{h+1} before B_{h+2}, it must receive a binary output from DBA_{h+1}. According to Lemma 2, this binary output must be 0, which directs p_j to commit B. □

APPENDIX C
ADDITIONAL RELATED WORK

In this section, we summarize the additional related works except in Section VII, specifically including the synchronous BFT consensus and partially-synchronous BFT consensus.

1) Synchronous BFT consensus: Synchronous BFT consensus protocols are designed under the network assumption that each message can be delivered within a predefined period, denoted as Δ, after its transmission. Representatives in this category encompass many early works [44], [45], [46] as well as some recent studies [47], [48]. However, protocols designed for synchronous networks encounter a challenge in setting the right value for Δ. If Δ is set too small, the synchronous assumption becomes fragile. Conversely, if Δ is set too large, the resulting protocol will be slow, as its performance must directly depend on Δ [48].

2) Partially-synchronous BFT consensus: Given the FLP impossibility [11], which states that deterministic fault-tolerant asynchronous consensus is impossible, Dwork et al. propose an intermediate network assumption called partial synchrony [49]. The partial synchrony model assumes the network to be synchronous after an unknown *Global Stabilization Time* (GST), which has been the mainstream model for practical systems for a long time.

One of the most notable works adopting the partially-synchronous assumption is PBFT [12]. Building on PBFT, subsequent works aim to reduce consensus latency by introducing a fast committing path [50], [51], [52], [53]. Drawing inspiration from the flourishing blockchain technology [2], structures like blocks and chains are incorporated into BFT consensus to pipeline consecutive consensus instances, thereby enhancing throughput. Example chained BFT consensus include Tendermint [54], Casper [55], and HotStuff [13]. Some works [56], [57] address liveness issues in chained-BFT where faulty leaders can prevent progress. Motorway constructs a data dissemination layer to improve throughput during periods of bad networks [58]. Amiri et al. propose a unified platform named Bedrock for partially-synchronous BFT protocols analysis, implementation, and experimentation [59].

Despite its popularity, the partially-synchronous protocols have raised concerns about their robustness [10], [7]. An adversary with network manipulation capacities can compromise the liveness of a partially-synchronous protocol. Consequently, a recent line of work is revisiting the asynchronous network in response to these concerns [34], [35], [36].

APPENDIX D
ARTIFACT APPENDIX

This appendix outlines the evaluation methodology for our artifacts. In Section VI, we present experimental results by deploying Ipotane on *Amazon Web Service* (AWS) with replicas distributed across five geographically dispersed regions: N.Virginia, Stockholm, Tokyo, Sydney, and N.California. Since configuring AWS involves a relatively complex process, we additionally provide local experimental instructions that can be executed on a single machine, thereby enabling easy validation of the code's functionality.

A. Description & Requirements

1) How to access: Source codes of Ipotane are available on Github[9], with a permanent archival record at Zenodo[10]. Detailed configuration and step-by-step execution instructions are available in the README.md file in the repository.

2) Hardware dependencies: No special hardware requirements are required.

3) Software dependencies: Ubuntu 22.04 LTS is recommended as the operating system. Other Linux distributions can technically support deployment, as long as the operator can complete the configuration of the environment dependencies. The runtime environment mandates the installation of Python 3.9 or later, Rust compiler version 1.50.0 or newer (with full toolchain support), Clang compiler (with standard library headers), and tmux terminal multiplexer for session management.

[9]https://github.com/CGCL-codes/ipotane
[10]https://doi.org/10.5281/zenodo.17008411

4) Benchmarks: We select 2-chain VABA, Ditto, Abraxas, and ParBFT as our benchmarks. Among these, 2-chain VABA[11] represents a purely asynchronous protocol, Ditto[11] exemplifies a serial dual-path asynchronous protocol, while both Abraxas[12] and ParBFT[13] serve as representatives of parallel dual-path asynchronous protocols.

B. Artifact Installation & Configuration

Our repository can be downloaded using the `git clone https://github.com/CGCL-codes/ipotane` command. We provide two ways to run our code: one is for local testing, and the other is for running on AWS.

1) Local testing: We offer two options for installing dependencies. The first is a 'dockerfile' that automatically generates a Docker image with all dependencies installed. The second is a manual installation method. To facilitate manual installation, we also provide a `build.sh` script. We recommend using the Docker approach for installation and execution. Detailed installation and configuration instructions will be provided in the *[Preparation]* part of Section D-D1.

2) Testing on AWS: After setting up AWS credentials and SSH keys, you can configure the environment by running commands such as `fab create` and `fab install`. For specific details, please refer to the *[Preparation]* part of Section D-D2.

C. Major Claims

- (C1): Ipotane is a novel dual-path asynchronous BFT consensus protocol that matches the performance of partially-synchronous protocols under favorable conditions while maintaining throughput and latency comparable to purely asynchronous protocols in unfavorable conditions.
- (C2): Under varying replica failure rates (ρ), Ipotane consistently achieves near-optimal performance. Specifically, when ρ=0, Ipotane performs on par with partially synchronous protocols, while at ρ=100%, its performance remains comparable to purely asynchronous protocols. These are supported by experiments, whose results are demonstrated in Section VI.B and VI.E, as well as Figures 6 and 9.
- (C3): In favorable conditions, Ipotane can continuously commit blocks by leveraging the 2-chain HotStuff protocol. In unfavorable conditions, it still achieves block commitment through consecutive DBA instances. This ensures stable throughput and latency across varying conditions. Experimental results, presented in Sections VI.C and VI.D as well as Figures 7 and 8, validate this performance.

D. Evaluation

In this section, we present the workflows for running Ipotane locally and on AWS.

[11]https://github.com/danielxiangzl/Ditto
[12]https://github.com/sochsenreither/abraxas
[13]https://github.com/ac-dcz/parbft-parbft1-rust

1) Local experiment process: Local deployment of Ipotane is relatively simple to implement.

[Preparation] There are two options to set up the testing environment.

Option 1: With Docker (Recommended). After changing to the project directory, execute the command below to build the Docker image, which has installed all dependencies required to run the experiment.

```
docker build -t ipotane
```

Then, execute the following command to launch a Docker container instance and enter its shell.

```
docker run -it ipotane /bin/bash
```

Option 2: Without Docker. You may choose to manually install the required dependencies, including:

- Rust 1.50.0+
- Python 3.9+
- tmux (for running processes in the background)
- Clang (dependency for RocksDB compilation)

For convenience, we include a `build.sh` script that automates the installation of all required dependencies.

[Execution] After successfully launching the Docker container or completing the manual environment setup, you can now perform the following operations:

```
git clone https://github.com/CGCL-codes/
ipotane
cd ipotane && cargo build
cd benchmark
pip install -r requirements.txt
```

These commands serve to clone the repository and install the required Python libraries. Note that the initial `cargo build` execution may take considerable time, as our implementation utilizes RocksDB—which requires compilation during this step.

To run the system, execute the `fab local` command within the ipotane/benchmark directory. The benchmark parameters can be customized in `fabfile.py`. Key configuration categories include:

[Benchmark parameters (bench_params)]

- `nodes`: number of replicas to run (default: 4)
- `duration`: test duration in seconds (default: 30)

[Node parameters (node_params)]

- `random_ddos`: whether to mount random DDos attacks on the leaders (default: False)
- `random_ddos_chance`: the probability of mounting random DDos attacks (default: 0)

[Results] When the `fab local` command completes, it displays an execution summary in the console and automatically saves detailed logs to the `logs` directory. You can use `fab logs` to parse these logs again, generating formatted results that match the console output and are saved to the `results` directory.

Using the default parameters described above, the Ipotane system will run locally with four replicas deployed on a single machine. These replicas benefit from an optimized network environment, resulting in significantly reduced latency measurements. This differs from the results reported in Section VI of our paper, which were obtained under *Wide Area Network* (WAN) conditions.

2) AWS-Based experiment process: The key difference between AWS and local deployments of Ipotane is in the preparation phase.

[Preparation] To deploy Ipotane on AWS, the following configuration steps must first be completed to set up the experimental environment.

- **Configure AWS credentials.** Enable programmatic access to your AWS account from your local machine. These credentials will authorize your system to programmatically create, modify, and delete EC2 instances.
- **Add SSH public key.** Manually add your SSH public key to each AWS region you intend to use.
- **Testbed Configuration** The file `settings.json` located in `ipotane/benchmark` contains all the configuration parameters of the testbed to deploy.
- **Testbed configuration.** Modify the `settings.json` file located in ipotane/benchmark to configure your testbed parameters.
- **Testbed deployment.** Execute `fab create` to provision new AWS instances. The creation logic is defined in `fabfile.py` under the 'create' task.
- **Dependency installation.** Run `fab install` to: (1) clone the repository on remote instances, and (2) install Rust language prerequisites.

For routine maintenance:

- Use `fab stop` to gracefully shut down the testbed.
- Use `fab start` to restart the testbed without recreating instances.

[Execution] After setting up the testbed, execute the protocol on AWS instances by running `fab remote`.

[Results] The `fab remote` command automatically collects logs from all replicas, enabling the result aggregation and log analysis similar to the local experiment workflow.

E. Customization

In addition to the parameters mentioned in Section D (`nodes`, `duration`, `ddoS`, `random_ddos`), you can also modify other parameters in `fabfile.py`, including:

- `tx_size`: transaction size in bytes (default: 512)
- `rate`: transactions input per second (default: 10,000)
- `faults`: Byzantine replicas to simulate (default: 0)