Aether: Approaching the Holy Grail in Asynchronous BFT

Xiaohai Dai[*], *Chaozheng Ding*[*], *Julian Loss*[†], *and Ling Ren*[‡]

[*]*Huazhong University of Science and Technology*

[†]*CISPA Helmholtz Center for Information Security*

[‡]*University of Illinois at Urbana-Champaign*

{xhdai, chaozhengding}@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 holy grail 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.

To approach our holy grail, we propose Aether, which specifically attains performance on par with purely asynchronous protocols in unfavorable situations—in both throughput *and* latency. Aether also runs two paths simultaneously: two-chain HotStuff as the optimistic path and a new primitive *Dual-functional Byzantine Agreement* (DBA) for the pessimistic path. DBA packs functionalities of biased ABA and *Validated Asynchronous Byzantine Agreement* (VABA). In Aether, each replica inputs 1 to DBA if its pessimistic path is faster, and 0 otherwise. DBA's ABA functionality promptly signals the optimistic path's failure by outputting 1, ensuring Aether's low latency in unfavorable situations. Meanwhile, Aether executes DBA instances to continuously produce pessimistic blocks through their VABA functionality. Upon detecting a failure, Aether commits the last two pessimistic blocks to maintain high throughput. Besides, Aether leverages DBA's biased property to ensure the safety of committing pessimistic blocks. Extensive experiments validate Aether's high throughput and low latency across all situations.

1 Introduction

The explosive popularity of blockchain technology [9, 58] has reignited significant interest in *Byzantine Fault Tolerant* (BFT) consensus over the past decade [53, 54]. At its core, BFT consensus empowers distributed replicas to reach an agreement on data, 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 [6, 20, 55] ensure safety and liveness under arbitrary network conditions, whereas (partially-)synchronous protocols are prone to network attacks [43]. On the flip side, asynchronous protocols are known to be inherently randomized [24], which makes them less efficient and more challenging to design than their synchronous and partially-synchronous counterparts (e.g., PBFT [15] and HotStuff [56] which can be fully deterministic).

1.1 Asyn. 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 [11, 17, 27, 41]. This typically involves using a partially-synchronous protocol, like two-chain HotStuff [34], 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. On the contrary, 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.

The holy grail in this dual-path paradigm is to match the performance of a partially-synchronous protocol in favorable situations and the performance of a purely asynchronous protocol in unfavorable ones. It is critical to optimize performance in unfavorable situations, as they can be common

in practice. Specifically, a leader may become temporarily inoperative, or the network connecting to the leader might experience jitter, making such 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 [27] and BDT [41] 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 6.

To deal with issues of sequential-path protocols, two recent works ParBFT [17][1] and Abraxas [11], 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.

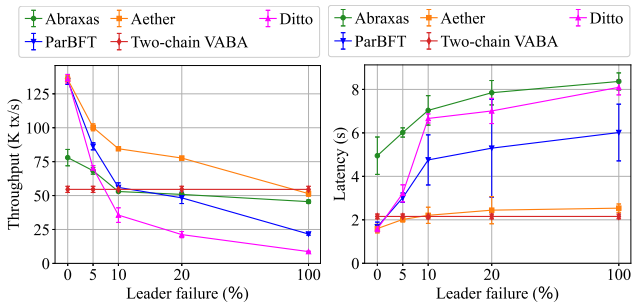
In spite of these improvements, ParBFT and Abraxas still fall short of fully matching the performance of asynchronous protocols in unfavorable situations. Concretely, ParBFT employs an individual *Asynchronous Binary Agreement* (ABA) instance at each height to detect the optimistic path's failure. This 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 a special indicator transaction on the pessimistic path confirms the optimistic path's failure. This indicator transaction is submitted after many communication rounds (empirically, more than 80 rounds). Thus, though Abraxas achieves high throughput *on average*, it might incur very high latency *in the worst case*.

Therefore, we pose the following natural question: *Is there 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?*

1.2 Our solution

We answer this question affirmatively by proposing Aether. Aether combines the advantages of Abraxas and ParBFT.

[1] In [17], two versions of ParBFT are proposed. Our focus is on the first one, ParBFT1, which we simply refer to as ParBFT in this paper.



(a) Throughput comparison (b) Latency comparison

Figure 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.

More precisely, Aether delivers performance akin to that of partially-synchronous protocols under favorable situations; in unfavorable situations, Aether still offers throughput and latency comparable to those of a purely asynchronous protocol. In addition, in situations that fall between favorable and unfavorable, Aether consistently maintains nearly the best throughput and latency among existing protocols.

At a high level, Aether executes the optimistic and pessimistic paths in parallel, much like Abraxas and ParBFT. The optimistic path runs the two-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. Besides, the two-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 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. We note that although DBA bears some similarity to Cachin et al.'s VABA [14], they actually differ significantly, as will be discussed in Section 3.2.2.

With the DBA primitive defined, Aether 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, Aether promptly commits blocks on the pessimistic path, thus

Table 1: Performance comparison. δ and Δ denote the actual network delay and timer parameter. c and L represent the maximum transaction count and block size of a block. κ denotes the computational security parameter, while λ is the lookback parameter in Abraxas. Performance in unfavorable situations is measured assuming the adversary mounts arbitrary attacks.

	Favorable situations			Unfavorable situations		
	Latency	Throughput	Communication	Latency	Throughput	Communication
Two-chain HotStuff [27]	5δ	$c/(2\delta)$	$O(nL+n\kappa)$	/	/	/
Two-chain VABA [27]	10.5δ	$2c/(7\delta)$	$O(n^2L+n^2\kappa)$	10.5δ	$c/(7\delta)$	$O(n^2L+n^2\kappa)$
Ditto [27]	5δ	$c/(2\delta)$	$O(nL+n\kappa)$	$3\Delta+10.5\delta$	$c/(2\Delta+7\delta)$	$O(n^2L+n^2\kappa)$
Abraxas [11]	5δ	$c/(2\delta)$	$O(n^2L+n^2\kappa)$	$3.5\lambda\delta+14\delta$ ‡	$c/(7\delta)$	$O(n^2L+n^2\kappa)$
ParBFT [17]	5δ	$c/(2\delta)$	$O(n^2L+n^2\kappa)$	22δ	$c/(22\delta)$	$O(n^2L+n^2\kappa)$
Aether	5δ	$c/(2\delta)$	$O(n^2L+n^2\kappa)$	18.5δ	$3c/(23\delta)$	$O(n^2L+n^2\kappa)$

† As a common implementation practice, transactions are constantly packaged and broadcast through an underlying mempool [27]. A consensus block only contains hashes of some transaction packages. Therefore, L is typically pretty small, approximately κ, which should not negatively impact the performance a lot [11].

‡ 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 [11].

promising low latency in unfavorable situations. Thanks 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, helping Aether achieve good throughput under unfavorable situations.

Experimental results to evaluate Aether 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 ($\rho = 0$), Aether 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 ($\rho = 100\%$), Aether demonstrates throughput and latency on par with two-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, Aether consistently achieves almost the highest throughput and lowest latency compared to other protocols.

Table 1 presents a more detailed and comprehensive comparison between Aether and existing protocols, corroborating Aether's good performance with theoretical analysis. δ denotes the actual network delay, and c represents the maximum transaction count within a block. Aether 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, two-chain HotStuff). On the other hand, in unfavorable situations, Aether manages to maintain a latency of 18.5δ and a throughput of $3c/(23\delta)$. These are just slightly worse than a purely asynchronous protocol (specifically, two-chain VABA) but significantly better than prior works Abraxas in terms of latency and ParBFT in terms of throughput.

2 Models And Preliminaries

2.1 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 posterior. 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 [8, 39]: 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.

2.2 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 mul-

tiple transactions. Transactions are continuously generated by clients or upper-layer applications, and are inserted into a buffer of each replica i, denoted as buf_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, denoted $\mathsf{buf}_i[:c]$.

$\mathcal{C}_i$ 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:

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 each index k ($k \geq 1$) and each non-faulty replica p_i, either buf_i remains forever empty or eventually $\mathcal{C}_i[k] \neq \perp$.*

2.3 (Biased) ABA

The *Asynchronous Binary Agreement* (ABA) abstraction [10, 25] 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 2. *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.*

The ABA protocol can be adapted to exhibit a bias towards 0 by replacing 'validity' property with a 'biased validity' property, which 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.

An ABA protocol achieving the biased-validity property is termed a biased ABA. 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 3.

2.4 VABA

Different from ABA, the *Validated Asynchronous Byzantine Agreement* (VABA) abstraction facilitates consensus on arbitrary values [14]. VABA introduces an external validation predicate Q, typically defined by higher-layer applications. To be more specific, VABA is defined as follows.

Definition 3. *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 [2, 14, 31, 42] have been developed over the past decades. Note that while the quality property is not explicitly defined in [14, 31], both works guarantee this property.

3 Building Block: DBA

3.1 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 distinct validity property 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.*
- ***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 $\langle 0, *\rangle$.*
- ***Proof 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. It is important to note that the block inputs of different replicas, like the input values in VABA, do not have to be identical. The properties of biased validity and proof validity specifically pertain to DBA's binary values, while quality and external validity are applicable to the block values. Besides, DBA discards the validity property defined in the original (biased) ABA abstraction. This implies that even if all non-faulty replicas input a binary value of 1 but some faulty replica inputs 0, DBA can output 0.

3.2 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 an existing 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.

3.2.1 AlgDBA protocol

This protocol, given in Algorithm 1, adds a communication round to amplify the bit of 0, prior to executing VABA. In this 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 in Algorithm 1). 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.

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

```
1  Let ⟨b_i, σ_i, B_i⟩ denote the input of p_i.
2  if b_i = 0 then
3  |  broadcast (h, 0, σ_i, SignShr_{t+1}(h, 0))
4  else
5  |  broadcast (h, 1, σ_i, SignShr_{n−t}(h, 1))
6  on receiving (h, 0, σ_j, *) s.t. P(σ_j) = True from p_j:
   |  // amplify the bit of 0
7  |  if p_i has not broadcast 0 then
8  |  |  broadcast (h, 0, σ_j, SignShr_{t+1}(h, 0))
9  on receiving t + 1 (h, 0, σ_j, *) that P(σ_j) = True:
10 |  S ← all the sig. shares from t + 1 messages;
11 |  sig_0 ← Comb_{t+1}(h, 0, S);
12 |  input ⟨0, sig_0, B_i⟩ to VABA_h if it has not done
13 on receiving n − t (h, 1, *, *):
14 |  S ← all the sig. shares from n − t messages;
15 |  sig_1 ← Comb_{n−t}(h, 1, S);
16 |  input ⟨1, sig_1, B_i⟩ to VABA_h if it has not done
17 on outputting ⟨b, sig, B⟩ from VABA_h:
18 |  output ⟨b, B⟩
```

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). It is important to note that 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.

3.2.2 Relation to Cachin et al. [14]

Cachin et al. [14] first introduced VABA and its variant, biased binary VABA. Biased binary VABA is very similar to biased ABA, as it also restricts the protocol to binary values with biased validity. While we draw inspiration from their work, our work is significantly different from theirs. The end goal of Cachin et al. [14] is to construct VABA that agrees on a block

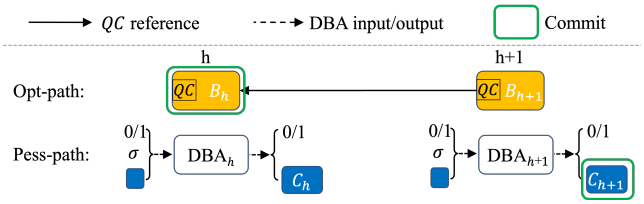


Figure 2: The structure of an epoch in Aether.

value, and they defined and used the biased binary variant 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.

4 Aether Design

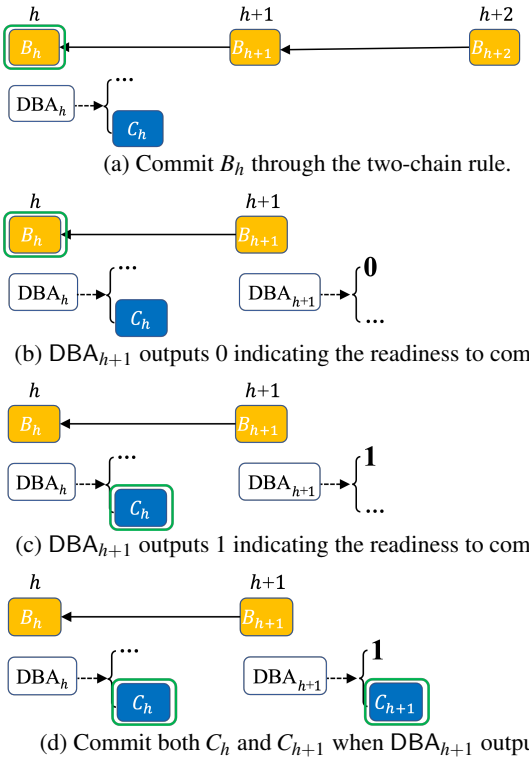
In this section, we present the design of Aether. Due to space limitations, the theoretical analysis, including both correctness and performance analysis, is provided in Appendix B.

4.1 Overview and intuition

Aether 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.

4.1.1 Design intuition

In the context of Aether, 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 $\mathcal{C}$.



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

(b) DBA_{h+1} outputs 0 indicating the readiness to commit B_h.

(c) DBA_{h+1} outputs 1 indicating the readiness to commit C_h.

(d) Commit both C_h and C_{h+1} when DBA_{h+1} outputs 1.

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

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 augmented VABA to DBA to make a binary decision. In particular, we leverage the binary output from the DBA instance **at the next height**, namely 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, Aether must be capable of rapidly committing blocks through the optimistic path, particularly employing the two-chain rule akin to two-chain HotStuff [27]. 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.

4.1.2 Overall design

Each replica participates in both the optimistic and pessimistic paths. The optimistic path, resembling the two-chain HotStuff, involves designated leaders proposing opt-blocks, which are then voted 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 a DBA instance. Therefore, 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.

Besides, 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 3d, both C_h and C_{h+1} are committed when DBA_{h+1} outputs 1.

In favorable situations, Aether continuously commits blocks through the two-chain rule, achieving performance akin to partially-synchronous protocols. In contrast, under unfavorable situations, Aether remains capable of committing 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 Aether to match performance of purely asynchronous protocols in unfavorable situations.

4.2 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 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 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 number h, and "DBA_h outputs O" implies O has a height number h.

We also define some functions utilized in Aether, including *GenOptBlk*, *InvokeDBA*, and *Commit*. Both *GenOptBlk* and *InvokeDBA* need to extract a batch of transactions from the replica's transaction buffer buf, which is achieved by calling the *GenTxBatch* function.

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
```

4.3 Detailed design when $h > 1$

Algorithm 3 outlines an epoch in Aether, which operates in consecutive heights[2]. This subsection describes the general protocol for heights greater than 1, with special considerations

[2]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.

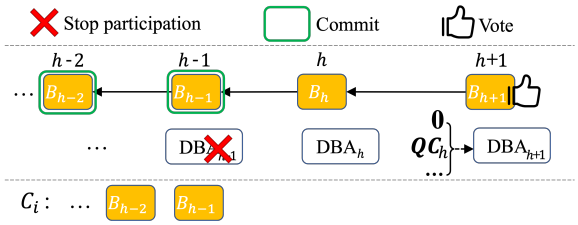


Figure 4: Actions taken when $t_1 < t_2$ in Aether.

for the first height discussed in the next subsection.

The 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 at height h, namely 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. p_i takes different actions based on this comparison.

4.3.1 Case 1: $t_1 < t_2$

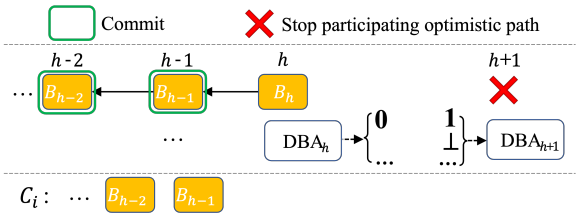
In this case, p_i receives B_{h+1} earlier than the output from DBA_h, indicating that the optimistic path works well as expected. p_i leverages the two-chain rule to commit block B_{h-1} (when $h \geq 2$) and casts a vote for the received block B_{h+1}, as described in Figure 4 and Lines 15-16 in Algorithm 3. Additionally, p_i stops participating in the DBA_{h-1} (when $h \geq 2$) instance (Line 17). Besides, 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. Besides, p_i will also broadcast B_{h+1} to make sure other replicas receive this block (Line 19).

Algorithm 3: An epoch in Aether for p_i

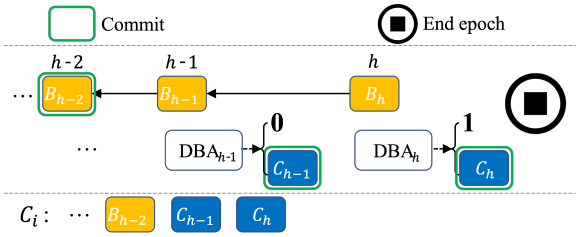
```
1  Let L_h denote the leader of height h on opt. path.
2  h ← 1, prevPessBlk ← ⊥.
   // optimistic path
3  if p_i is L_1 then
4  |  B_1 ← GenOptBlk(1, ⊥); broadcast B_1
   // pessimistic path
5  InvokeDBA(1, 0, ⊥)
   // optimistic path
6  on receiving B_1:
7  |  send SignShr_{n-t}(B_1) to L_2
8  on receiving n-t sign. shares on B_k (denoted as S):
9  |  if p_i is L_{k+1} then
10 |  |  qc ← Comb_{n-t}(B_k, S); B_{k+1} ← GenOptBlk(k+1, qc);
11 |  |  broadcast B_{k+1}
12 while the epoch is not concluded:
13 |  wait until B_{h+1} is received or DBA_h outputs O
14 |  if B_{h+1} is received earlier then
   |  |  // optimistic path
15 |  |  Commit(B_{h-1}) if h ≥ 2;
16 |  |  send SignShr_{n-t}(B_{h+1}) to L_{h+2};
17 |  |  stop participating in DBA_{h-1} if h ≥ 2;
18 |  |  InvokeDBA(h+1, 0, B_{h+1}.QC);
19 |  |  broadcast B_{h+1} if it has not done yet
20 |  else
   |  |  // pessimistic path
21 |  |  if O.b = 0 then
22 |  |  |  stop participating in the optimistic path;
23 |  |  |  InvokeDBA(h+1, 1, ⊥);
24 |  |  |  if h ≥ 2 and B_{h-1} isn't committed then
25 |  |  |  |  wait to receive B_{h-1} and Commit(B_{h-1});
26 |  |  |  prevPessBlk ← O.C
27 |  |  else
28 |  |  |  Commit(prevPessBlk); Commit(O.C);
29 |  |  |  conclude the epoch
30 |  h ← h+1
```

4.3.2 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, namely 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



(a) DBA_h outputs 0.



(b) DBA_h outputs 1.

Figure 5: Actions taken when $t_1 \geq t_2$ in Aether.

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 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 at height $h-1$ (when $h \geq 2$), namely B_{h-1}. 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 outputted from DBA_h, namely C_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 the current DBA, namely DBA_h. Subsequently, p_i concludes its participation in the current epoch and progresses to the next epoch.

4.4 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 two-chain HotStuff [27]. 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 DBA_1 instance will straightforwardly recognize this binary input of 0 as valid.

5 Implementation and Evaluation

In this section, we present the implementation of Aether and conduct a comparison with other protocols. Our chosen baselines include Abraxas and ParBFT, both of which employ the parallel-path paradigm similar to Aether. 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 two-chain VABA, a purely asynchronous protocol, as a baseline for the evaluation of unfavorable situations.

5.1 Implementation and experimental setup

5.1.1 Implementation

We directly adopt the available open-source codes of our baselines ParBFT[3] and Abraxas[4]. Two-chain VABA and Ditto share the same repository[5]. 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 payloads, effectively reducing the size of consensus messages and enhancing performance.

To ensure a fair comparison, we implement Aether using the same framework as the baselines. The VABA protocol employed in DBA is instantiated with sMVBA [31]. 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 two-chain VABA [27]. Besides, the view-change phase has been streamlined to a single round following AMS-VABA [4].

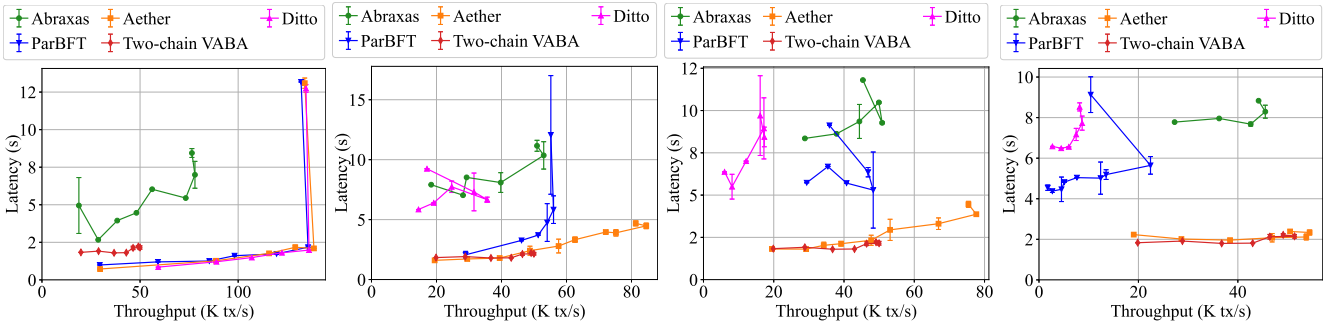
5.1.2 Experimental setup

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. In Ditto, the timing parameter Δ is configured to 5 seconds, while the lookback parameter λ in Abraxas is fixed at 20.

Except for two-chain VABA, each protocol employs predetermined leaders for optimistic paths. Depending on leader crash frequency, we consider 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 two-chain VABA,

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



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

Figure 6: Latency vs. throughput.

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.

Our experiments are conducted on AWS, where each replica is deployed as an m5d.2xlarge instance. Each instance is equipped with 8 vCPUs and 32GiB memory, running Ubuntu 20.04 as the operating system. 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.

5.1.3 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.

5.2 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.

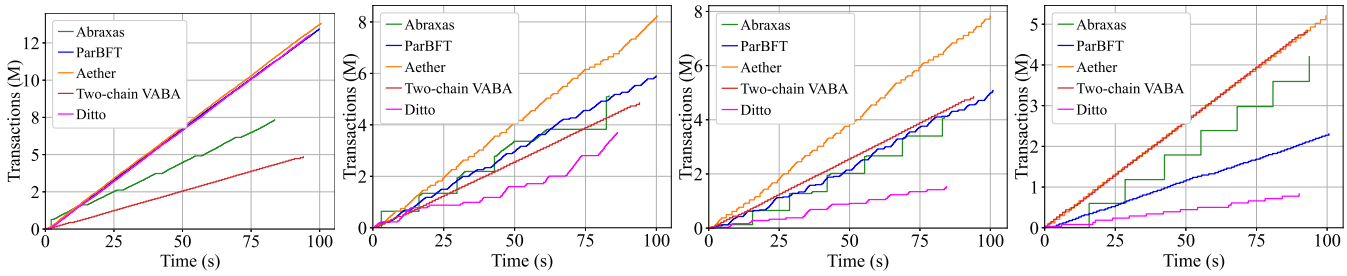
As shown in Figure 6a, when the optimistic path always operates well, Aether attains low latency and high throughput, comparable to Ditto and ParBFT.[6] Notably, Ditto matches a partially-synchronous protocol's performance, as it adopts the sequential-path paradigm and only runs the optimistic path in this scenario. Thus, in favorable situations, Aether'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, Aether still maintains good performance, slightly inferior to the purely asynchronous protocol (two-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, Aether 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, Aether consistently outperforms others as illustrated in Figure 6b and Figure 6c. In terms of latency, Aether and two-chain VABA consistently demonstrate the lowest among these protocols.

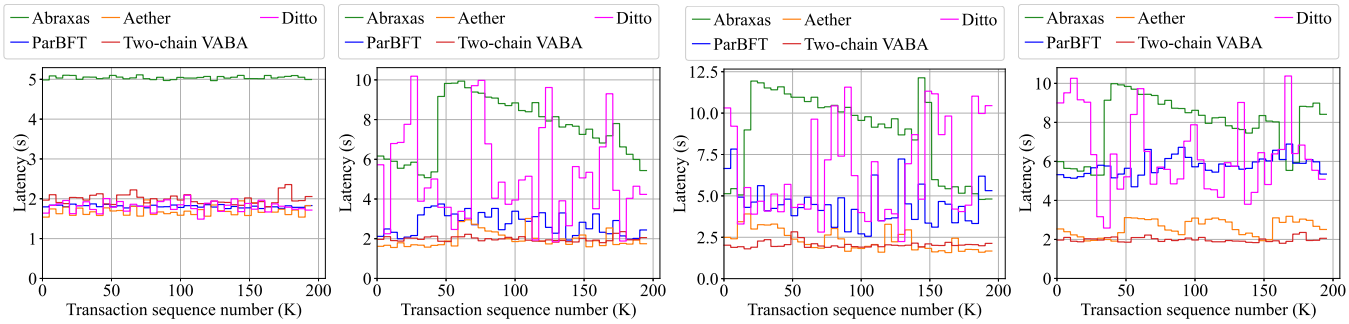
To sum up, across all scenarios, Aether consistently attains the (near-)best performance among all protocols. Specifically, it achieves performance on par with partially-synchronous protocols in favorable situations and on par with purely asynchronous ones in unfavorable situations.

[6] 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) $\rho = 100\%$

Figure 7: Throughput over time.



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

Figure 8: Latency over the transaction sequence numbers.

5.3 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 two-chain VABA protocol 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, two-chain VABA, ParBFT, and Aether 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.

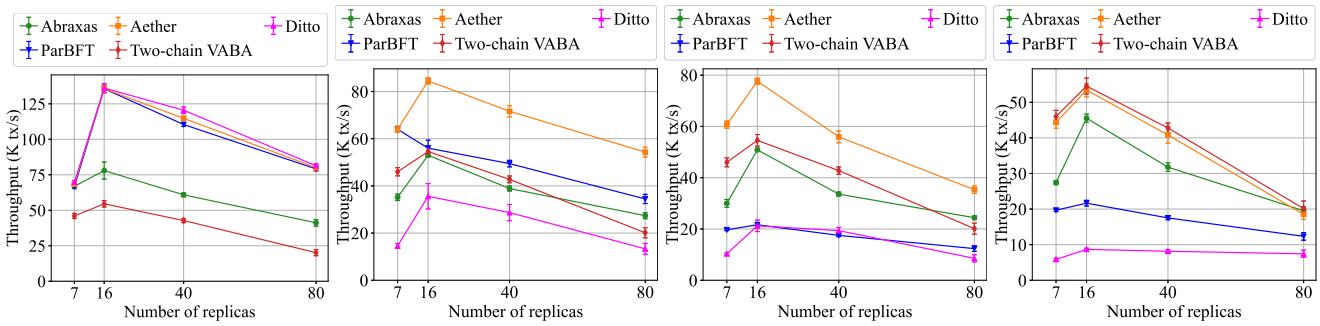
In the $\rho = 10\%$ or $\rho = 20\%$ scenario, all protocols except VABA exhibit less stable throughput as they alternate between committing blocks through the optimistic path and the pessimistic path. Nevertheless, Aether continues to showcase superior stability than Abraxas and Ditto.

5.4 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), Aether maintains relatively stable latency by committing pess-blocks through successive DBA instances. While Aether's latency deviation is slightly larger than that of two-chain VABA, it is more stable than others. Aether'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, Aether and VABA still maintain stable latency, with fluctuation significantly lower than other protocols.



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

Figure 9: Throughput vs. system size.

5.5 Scalability evaluation

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

As depicted in Figure 9a, Aether, 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, Aether consistently maintains high throughput comparable to two-chain VABA, across different replica counts. When leaders on the optimistic path fail with a probability of 10% or 20%, Aether outperforms all other protocols under varying system sizes, as evidenced by Figure 9b or Figure 9c. In summary, Aether consistently demonstrates excellent scalability across diverse probabilities of leader failures.

6 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 [1,44,48,52]. VABA and MVBA instead focus on agreeing on arbitrary values [2,14,26, 42]. Building upon ABA or VABA, *Asynchronous Common Subset* (ACS) and SMR can be constructed [21,32,43,57].

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 path in sequence [7,37,49], necessitating path switches [27,41]. 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 [11] and ParBFT [17], 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. Aether proposed in this paper achieves both high throughput and low latency in both favorable and unfavorable situations.

Another class of protocols [18,35,50] 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 compared to many previously discussed protocols. Besides, these approaches generally depend on multiple rounds of *Reliable Broadcast* (RBC) to achieve consensus, resulting in high latency. For instance, DAGRider and Tusk require latencies of 12δ and 9δ, respectively, even under favorable situations. BullShark [51], 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, slightly larger than the 5δ offered by a partially-synchronous protocol (e.g., two-chain HotStuff) or our Aether. 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 the 10.5δ typical of purely asynchronous protocols (e.g., two-chain VABA) or 18.5δ offered by Aether.

7 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 Aether, 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, Aether leverages DBA instances to continuously produce blocks without idle periods, thereby achieving high throughput in unfavorable situations. In summary, Aether 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.

References

[1] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 381–391. ACM, 2022.

[2] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling 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*, pages 320–334. Springer, 2019.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 106–118. IEEE, 2020.

[4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346. ACM, 2019.

[5] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The bedrock of byzantine fault tolerance: A unified platform for {BFT} protocols analysis, implementation, and experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 371–400, 2024.

[6] Diogo S Antunes, Afonso N Oliveira, André Breda, Matheus Guilherme Franco, Henrique Moniz, and Rodrigo Rodrigues. {Alea-BFT}: Practical asynchronous byzantine fault tolerance. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 313–328, 2024.

[7] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems*, 32(4):1–45, 2015.

[8] Renas Bacho and Julian Loss. On the adaptive security of the threshold bls signature scheme. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 193–207, 2022.

[9] Imran Bashir. *Mastering Blockchain*. Packt Publishing Ltd, 2017.

[10] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30. ACM, 1983.

[11] Erica Blum, Jonathan Katz, Julian Loss, Kartik Nayak, and Simon Ochsenreither. Abraxas: Throughput-efficient hybrid asynchronous consensus. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, page 519–533. ACM, 2023.

[12] Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016.

[13] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 2001 Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 1999 USENIX Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX, 1999.

[16] Xiaohai Dai, Liping Huang, Jiang Xiao, Zhaonan Zhang, Xia Xie, and Hai Jin. Trebiz: Byzantine fault tolerance with byzantine merchants. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 923–935. ACM, 2022.

[17] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. Parbft: Faster asynchronous bft consensus with a parallel optimistic path. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, page 504–518. ACM, 2023.

[18] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the 17th European Conference on Computer Systems*, pages 34–50. ACM, 2022.

[19] Danny Dolev, Michael J Fischer, Rob Fowler, Nancy A Lynch, and H Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

[20] Sisi Duan, Michael K. Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041. ACM, 2018.

[21] Sisi Duan, Xin Wang, and Haibin Zhang. Fin: Practical signature-free asynchronous common subset in constant time. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 815–829, 2023.

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

[23] M Fischer, R Fowler, and N Lynch. A simple and efficient byzantine generals algorithm. In *Proceedings, Second Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh*, 1982.

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

[25] Roy Friedman, Achour Mostefaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, 2005.

[26] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1187–1201. ACM, 2022.

[27] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Proceedings of the 2022 International Conference on Financial Cryptography and Data Security*, pages 296–315. Springer, 2022.

[28] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Motorway: Seamless high speed bft. *arXiv preprint arXiv:2401.10369*, 2024.

[29] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. Beegees: stayin'alive in chained bft. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 233–243, 2023.

[30] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 568–580. IEEE, 2019.

[31] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. *Cryptology ePrint Archive*, 2022.

[32] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818. ACM, 2020.

[33] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[34] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and robust bft protocol for blockchains. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[35] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175. ACM, 2021.

[36] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58. ACM, 2007.

[37] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, pages 204–215. Springer, 2005.

[38] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[39] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 303–312, 2014.

[40] Jian Liu, Wenting Li, Ghassan O. Karame, and Nadarajah Asokan. Scalable byzantine consensus via hardware-assisted secret sharing. *IEEE Transactions on Computers*, 68(1):139–151, 2018.

[41] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2159–2173. ACM, 2022.

[42] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing*, pages 129–138. ACM, 2020.

[43] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[44] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t< n/3 and o (n2) messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 2–9. ACM, 2014.

[45] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[46] Jianyu Niu, Fangyu Gai, Mohammad M Jalalzai, and Chen Feng. On the performance of pipelined hotstuff. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[47] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[48] Michael O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 403–409. IEEE, 1983.

[49] HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Proceedings of the 2005 International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

[50] Maria A. Schett and George Danezis. Embedding a deterministic bft protocol in a block dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 177–186. ACM, 2021.

[51] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718. ACM, 2022.

[52] Sam Toueg. Randomized byzantine agreements. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 163–178. ACM, 1984.

[53] Xin Wang, Sisi Duan, James Clavin, and Haibin Zhang. Bft in blockchains: From protocols to use cases. *ACM Computing Surveys*, 54(10):1–37, 2022.

[54] Yang Xiao, Ning Zhang, Wenjing Lou, and Y. Thomas Hou. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials*, 22(2):1432–1465, 2020.

[55] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, 2022.

[56] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019.

[57] Haibin Zhang and Sisi Duan. Pace: Fully parallelizable bft from reproposable byzantine agreement. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3151–3164, 2022.

[58] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018.

A Correctness Analysis of AlgDBA

In this section, we prove our AlgDBA construction adheres to all the properties outlined in Section 3.1.

A.0.1 Agreement, quality, and external validity

These properties of AlgDBA are derived directly from VABA.

A.0.2 Termination

AlgDBA's termination property is stated in Theorem 2, supported by Lemma 1.

LEMMA 1. *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 2. AlgDBA *achieves termination.*

Proof. Based on Lemma 1, 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. □

A.0.3 Proof 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.

A.0.4 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.

B Analysis of Aether

Our analysis of Aether covers two main aspects: correctness and efficiency. Correctness analysis examines whether Aether fulfills SMR's three properties, namely consistency, liveness, and completeness.

B.1 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 8 addresses the consistency property, supported by Lemmas 3, 4, 5, 6, and 7.

LEMMA 3. *If a non-faulty replica concludes an epoch in the 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 4. *Within an epoch, if a non-faulty replica commits an opt-block at height h and another non-faulty replica output 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 4.3. 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 5. *Within an epoch, if two non-faulty replicas commit two blocks at the same height, then either both two 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 4, p_j will receive 0 from DBA_{h+1} if it receives an output at all. According to the protocol described in Section 4.3, 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 3, 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 6. *Within an epoch, if two non-faulty replicas commit two blocks at the same height, these two blocks must be identical.*

Proof. Per Lemma 5, 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 4.3, 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 proof validity property, a replica must have input a valid tuple $\langle 0, \sigma, * \rangle$ 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 7. *If two non-faulty replicas conclude the same epoch, they must commit the same number of blocks within that epoch.*

Proof. Based on Lemma 3, 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. □

THEOREM 8 (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 7 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 6, $\mathcal{C}_i[k]$ and $\mathcal{C}_j[k]$ must be identical. □

B.2 Liveness analysis

We say the protocol concludes an epoch if any non-faulty replica concludes it. By Lemma 3, 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 2.2, 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 2.2, 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 that when tx's index becomes 0 or negative, tx is committed.

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 10, whose proof relies on Lemma 9.

LEMMA 9. *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 4, this binary output must be 0, which directs p_j to commit B. □

THEOREM 10 (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 9, 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 being 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 9, every non-faulty replica will also commit tx. □

B.3 Completeness Analysis

THEOREM 11 (COMPLETENESS). *For each index k ($k \geq 1$) and each non-faulty replica p_i, either* buf_i *remains forever empty or eventually $\mathcal{C}_i[k] \neq \perp$.*

Proof. If no new transactions are input by clients, buf_i will remain forever empty, thereby validating the theorem. Next, we consider the situation where transactions are continuously input by clients.

For a given transaction tx, according to the established liveness property, each non-faulty replica will eventually commit a block that contains tx. Consequently, when transactions are continuously input, each non-faulty replica will keep committing new blocks. Since blocks are committed sequentially, for every index k ($k \geq 1$), $\mathcal{C}_i[k]$ will eventually be non-empty.

To sum up, the completeness property is established. □

B.4 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. Besides, we assume the size of shares and signatures to all have length κ. Our analysis focuses on the efficiency of Aether when employing sMVBA [31] to construct AlgDBA. Inspired by AMS-VABA [4] and two-chain VABA [27], 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 [4], 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 Aether'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 two-chain VABA [27]. This way, committing a PB1-block in an AlgDBA will also commit a PB2-block from the preceding AlgDBA, thus improving DBA's throughput.

Favorable situation. In a favorable situation, 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 two 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, total communication overhead in a favorable situation is $O(n^2L + n^2\kappa)$.

Unfavorable situation. In an unfavorable situation, only two AlgDBA instances produce outputs per epoch. For simplicity, we refer to them 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 $O(n^2L + n^2\kappa)$.

C Additional Related Work

In this section, we summarize the additional related works except in Section 6, specifically including the synchronous BFT consensus and partially-synchronous BFT consensus.

C.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 [19, 23, 38, 47] as well as some recent studies [3, 33]. 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 [3].

C.2 Partially-synchronous BFT consensus

Given the FLP impossibility [24], which states that deterministic fault-tolerant asynchronous consensus is impossible, Dwork et al. propose an intermediate network assumption called partial synchrony [22]. 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 [15]. Building on PBFT, subsequent works aim to reduce consensus latency by introducing a fast committing path [16, 30, 36, 40]. Drawing inspiration from the flourishing blockchain technology [45], structures like blocks and chains are incorporated into BFT consensus to pipeline consecutive consensus instances, thereby enhancing throughput. Example chained BFT consensus include Tendermint [12], Casper [13], and HotStuff [56]. Some works [29, 46] address liveness issues in chained-BFT where faulty leaders can prevent progress. Besides, Motorway constructs a data dissemination layer to improve throughput during periods of bad networks [28]. Amiri et al. propose a unified platform named Bedrock for partially-synchronous BFT protocols analysis, implementation, and experimentation [5].

Despite its popularity, the partially-synchronous protocols have raised concerns about their robustness [20, 43]. 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 [21, 32, 57].