applied sciences

MDPI

Article

FlexBFT: A Flexible and Effective Optimistic Asynchronous BFT Protocol

Anping Song * and Cenhao Zhou

School of Computer Science and Engineering, Shanghai University, Shanghai 200444, China
* Correspondence: apsong@shu.edu.cn

Abstract: Currently, integrating partially synchronous Byzantine-fault-tolerant protocols into asynchronous protocols as fast lanes represents a trade-off between robustness and efficiency, a concept known as optimistic asynchronous protocols. Existing optimistic asynchronous protocols follow a fixed path order: they execute the fast lane first, switch to the slow lane after a timeout failure, and restart the fast lane after the slow lane execution is completed. However, when confronted with prolonged network fluctuations, this fixed path sequence results in frequent failures and fast lane switches, leading to overhead that diminishes the efficiency of optimistic asynchronous protocols compared with their asynchronous counterparts. In response to this challenge, this article introduces FlexBFT, a novel and flexible optimistic asynchronous consensus framework designed to significantly enhance overall consensus performance. The key innovation behind FlexBFT lies in the persistence of slow lanes. In the presence of persistent network latency, FlexBFT can continually operate round after round within the slow lane—the current optimal path—until the network conditions improve. Furthermore, FlexBFT offers the flexibility to combine consensus modules adaptively, further enhancing its performance. Particularly in challenging network conditions, FlexBFT's experimental outcomes highlight its superiority across a range of network scenarios compared with state-of-the-art algorithms. It achieves a performance with 31.6% lower latency than BDT, effectively merging the low latency characteristic of deterministic protocols with the robustness inherent in asynchronous protocols.

Keywords: blockchain; decentralization; Byzantine fault tolerance; asynchronous consensus; optimistic path

Citation: Song, A.; Zhou, C. FlexBFT: A Flexible and Effective Optimistic Asynchronous BFT Protocol. *Appl. Sci.* **2024**, *14*, 1461. https://doi.org/10.3390/app14041461

Academic Editor: Ana-Belén Gil-González

Received: 27 December 2023
Revised: 4 February 2024
Accepted: 7 February 2024
Published: 10 February 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Blockchain technology has garnered substantial attention within the field of distributed systems primarily for its inherent ability to ensure data consistency and resist unauthorized modifications [1]. In the realm of blockchain, state machine replication (SMR) plays a pivotal role in upholding ledger consistency among blockchain nodes. In this context, Byzantine fault tolerance (BFT) protocols, which are crucial for implementing SMR, have earned significant recognition for their robust fault-tolerant properties [2].

In scenarios involving both synchronous and partially synchronous networks, BFT protocols have witnessed remarkable advancements in terms of efficiency improvements [3–6]. They hold a central position in modern blockchain and distributed systems [7,8]. These algorithms typically feature a leader node working in conjunction with multiple replica nodes, and their innovative techniques, including threshold signatures and pipelining, have significantly reduced time and message complexities, resulting in linear time complexity.

Nonetheless, the groundbreaking work by Miller et al. [9] has shed light on a pivotal revelation: despite their remarkable efficiency in synchronous networks, these high-performance protocols face significant challenges when confronted with the unpredictable nature of asynchronous network conditions. This complex predicament leads to a standstill in the progress of state machine replication (SMR), rendering it incapable of seamless operation. This dilemma presents a formidable challenge for blockchain protocols, given that

real-world network environments often exhibit inherent asynchrony, be it in the realm of industrial blockchain deployments [10] or the intricate ecosystems of cryptocurrencies [11]. Consequently, the academic community has experienced a noticeable shift in focus toward asynchronous protocols, a subject that has been a continuous subject of scholarly inquiry since the 1980s.

In contrast with their synchronous counterparts, asynchronous protocols function under the assumption of message eventual delivery without strict emphasis on the precise timing of message arrivals. It is worth noting that while asynchronous protocols often incorporate randomness as an auxiliary computational resource, they are capable of ensuring correctness in asynchronous network environments, albeit sometimes at the expense of reduced efficiency compared with their synchronous counterparts during normal operational conditions.

To address the performance limitations of asynchronous protocols, researchers have recently introduced optimistic asynchronous protocols [12–15]. These algorithms typically incorporate two distinctive paths: an optimistic path, reliant on multicast communication, and a pessimistic path, founded on complete asynchronous consensus. The optimistic path leverages multicast to operate swiftly under favorable network conditions, delivering low latency and minimal communication overhead. Conversely, the pessimistic path, rooted in asynchronous protocols, offers robustness against foreseeable errors and challenges that might impede the fast lane.

While optimistic two-phase asynchronous protocols excel in synchronous networks with low complexity and offer reliability in asynchronous environments, they follow a fixed path order. In each round, the algorithm initially executes the optimistic path. In cases where the optimistic path fails to make progress, such as in asynchronous networks, it abandons this path and transitions to the pessimistic path to ensure consensus continuity. Following the completion of the pessimistic path, the algorithm recommences a new round of the fast lane as the entry point.

However, adhering to this fixed path order proves detrimental, especially in fully asynchronous networks or under latency attacks. Expanding the role of the fast lane in these optimistic asynchronous protocols results in additional overhead compared with pure asynchronous protocols. In adversarial networks, the fast lane, resembling semisynchronous consensus, rarely advances. Adversaries initiate failures, switch to the slow lane, and then encounter repeated failures in subsequent consensus rounds after producing output in the slow lane. Consequently, optimistic asynchronous consensus degenerates into complete asynchronous consensus, marked by the inefficiency of a failing fast lane and frequent switches, diminishing its effectiveness compared with pure asynchronous consensus.

In this paper, we introduce FlexBFT, a groundbreaking and adaptable optimistic asynchronous consensus framework poised to substantially enhance overall consensus performance. As Figure 1 shows, the pivotal innovation revolves around the sustained utilization of the slow lane, called back-up mode in FlexBFT. In this revamped approach, the slow lane serves not only as a fallback option in case of fast lane failure but also as an additional entry point for each consensus round (originally, only the fast lane is). In favorable network conditions, FlexBFT orchestrates consensus through its fast lane, also called fast mode in FlexBFT. However, when confronted with elevated network latency, FlexBFT gracefully transitions to the slow lane for multiple rounds. It persists in operating within the slow lane until it identifies an improvement in network conditions, at which point it reverts to the fast lane. This approach circumvents the burden of a faltering fast lane and the resource wastage associated with frequent switches in the presence of persistent asynchronous networks. Instead, it continuously conducts consensus optimally, namely, through the persistent execution of the slow lane. Furthermore, this paper offers the flexibility to integrate decoupled consensus modules, further enhancing performance with minimal associated costs.

Among the myriad use cases for blockchain, we highlight two scenarios particularly suited to FlexBFT. The first is the Internet of Things (IoT) context, where, with the evolu-

tion of IoT concepts, the decentralized management of IoT devices becomes increasingly logical. These devices are characterized by the potential need to process a large number of transactions and the unpredictability of communication stability. Thus, FlexBFT, capable of providing objective scalability in unstable networks, is well suited for IoT scenarios. The second scenario is cross-border trade, a realm where blockchain inherently fits well within financial contexts. The network of nodes spread across the globe in cross-border trade represents an asynchronous network, for which FlexBFT can offer a secure and efficient consensus process.

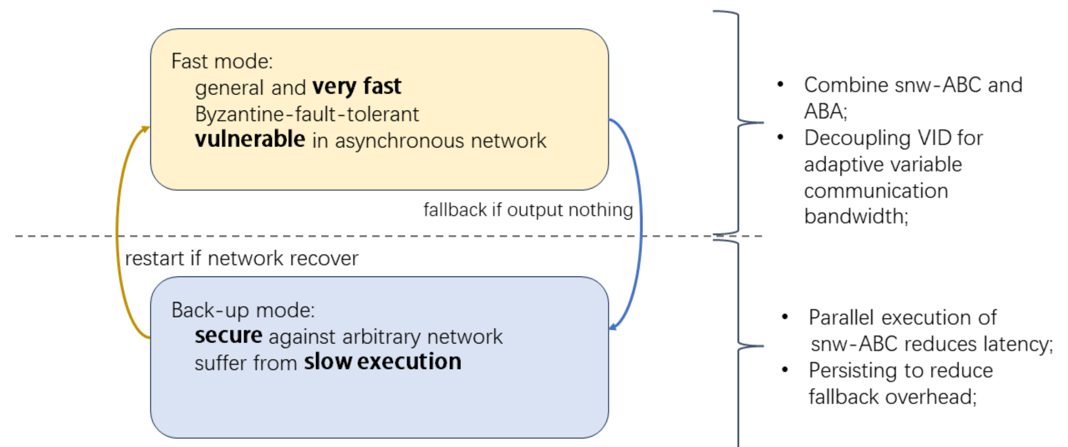

Figure 1. The overview of FlexBFT.

Our work in this paper encompasses the following significant contributions:

- Design of FlexBFT: We introduce FlexBFT, a highly adaptable optimistic asynchronous framework built upon Bolt-Dumbo Transformer (BDT) [14]. A standout feature of FlexBFT is its ability to persistently operate slow lanes, even in the presence of continuous asynchronous network conditions. Additionally, FlexBFT dynamically reactivates fast lanes when improvements in the network environment are detected. This flexible approach enables FlexBFT to effectively address diverse network scenarios, facilitating optimal consensus while reducing system overhead and latency.
- Flexible integration of consensus modules: We provide the flexibility to integrate decoupled consensus modules within FlexBFT, contributing to enhanced performance. These modules encompass the utilization of verified information delivery (VID) in the fast lane, achieving consensus prior to downloading, and concurrent execution of the fast lane within the slow lane in back-up mode to fully exploit available bandwidth resources. Our approach prioritizes efficient resource allocation by separating considerations for bandwidth resources and communication resources.
- Implementation and simulation: We have implemented FlexBFT using Python 3.7.4 and conducted comprehensive simulation experiments. Comparative analysis against BDT demonstrates that FlexBFT exhibits superior performance, particularly excelling in scenarios characterized by continuous network fluctuations.

2. Related Works

Based on different timing assumptions, BFT protocols can be classified into three categories: synchronous, partially synchronous, and asynchronous.

2.1. Synchronous and Semisynchronous BFT Protocols

The exploration of BFT consensus protocols traces back to the foundational work by Lamport et al [16]. Over the years, synchronous [17,18] and semisynchronous [19] BFT protocols came into existence, the majority of which were characterized by deterministic protocols. Among them, PBFT [3] stands as a classic example, establishing fundamental

concepts like the view-change mechanism and voting with message exchange. Subsequent work aimed to enhance PBFT for lower latency and higher throughput. HotStuff [6], an advanced BFT-like consensus algorithm, introduced a three-phase consensus incorporating threshold signatures for quorum certificates (QC) to replace multicast-based voting, reducing message complexity. Additionally, optimistic consensus protocols such as Zyzzyva [4], CheapBFT [20], SBFT [5], FastBFT [21], Flexico [22], etc., integrated optimistic fast paths, showcasing excellent performance in a fault-free scenario where all nodes operate as intended. Zyzzyva enhances performance through speculation, bypassing explicit consensus by executing requests as ordered by the primary. CheapBFT utilizes TEEs in its optimistic BFT protocol, requiring participation from only $f + 1$ replicas for execution and consensus. FastBFT, another TEE-based BFT method, secures consensus with $2f + 1$ replicas using secret sharing for security. SBFT combines fast, optimistic execution with linear PBFT for efficient dual-mode consensus. These protocols typically follow a fixed path order, primarily relying on the fast lane as an entry point, while semisynchronous BFT protocols are resorted to if and only if the fast path encounters failure. These synchronous and semisynchronous BFT protocols often exhibit lower latency but tend to lose liveness when confronted with an asynchronous network.

2.2. Asynchronous BFT Protocols

Research on asynchronous BFT protocols [23,23–26] also began early, but due to practical constraints, many of these investigations have remained theoretical. It was only relatively recently that HoneyBadgerBFT [9] was introduced. HoneyBadgerBFT employs reliable broadcast (RBC) and asynchronous binary agreement (ABA) to achieve asynchronous atomic broadcast (ABC). This approach has been continued in subsequent research, including BEAT [27], DispersedLedger [28], EPIC [29], etc. BEAT implements ABC with components of superior performance, offering multiple solutions tailored to different application domains and focal points. DispersedLedger introduces AVID-M, enhancing the parallelism of the ABC structure. EPIC focuses on addressing the liveness issue and achieves adaptive security through the practical application of Cobalt ABA. A different asynchronous consensus approach, exemplified by Dumbo [30], VABA [31], and Dory [32], employs MVBA to achieve asynchronous atomic broadcast. That is, Dumbo reduces the number of ABAs required for MVBA through a committee model. The basic abstractions such as RBC [33,34], ABA [35,36], and MVBA [30,31] have also seen continuous improvement. These asynchronous BFT protocols exhibit robustness, ensuring effectiveness even in asynchronous network environments. However, due to the constraints of the famous FLP impossibility states [37], asynchronous BFT protocols often require the introduction of randomness, resulting in higher latency.

2.3. Optimistic Asynchronous BFT Protocols

Optimistic asynchronous BFT protocols were introduced to address the efficiency limitations of asynchronous BFT. These protocols employ semisynchronous BFT as the primary, high-speed lane, while utilizing asynchronous BFT as the secondary, slower backup. Recent research contributions, exemplified by BDT [14] and Jolteon and Ditto [15], aim to optimize this approach. Gelashvili and colleagues [15] combined the deterministic HotStuff with a recently proposed MVBA [31] to achieve robust randomization-based view change in asynchronous network. Therefore, Ditto accomplishes handling failures in views caused by asynchronous networks or faults through MVBA, striving to minimize the overhead associated with switching during failures. Within BDT, transactions can pass through three modules sequentially: Bolt, representing the fast lane through deterministic protocols, Dumbo, functioning as the slow lane using MVBA, and Transformer, enabling asynchronous pace synchronization via low-cost ABA. In virtue of the adaptable yet resilient characteristics of the Blot–Transformer synergy within BDT, it exhibits exceptional performance in asynchronous network scenarios. Similar to semisynchronous optimistic BFT, these optimistic asynchronous BFT protocols follow a fixed path order, primarily

starting with the fast lane. Nevertheless, within the realm of asynchronous networks, the resource of communication subject to arbitrary delays assumes a critical value, thus rendering frequent transitions from the fast to the slow lane a substantial source of overhead, thereby providing impetus for our research endeavor.

3. Problem Formulation

3.1. System Model

We consider a standard asynchronous message-passing system with a trusted setup, as follows:

Trusted settings: Assume that there are n nodes, P_{1-n}, in the system. These n nodes know each other and the public keys of other nodes, and they all have correct cryptography settings so that hashing, encryption and decryption, and signature operations in the protocol are all correct. This can be considered as having a trusted third party during the system setup phase, or through distributed key generation [38].

Adversary limitations: The adversary's computing resources are limited and cannot find a pair of colliding hashes or crack asymmetric encryption in polynomial time.

Static Byzantine failure: The adversary is given full control over up to f failed nodes before the protocol begins, where f is a protocol parameter. In our protocol, the number of faulty nodes is limited to $n \geqslant 3f + 1$, and the faulty nodes controlled by the adversary are not allowed to change during the execution of the protocol.

Fully asynchronous network: Assume that each pair of nodes is connected by a reliable P2P channel, and the channel will not drop messages. Delivery time is determined by the adversary, but messages between the correct nodes must eventually be delivered.

Timeout mechanism: In the context of asynchronous networks, the conventional determination of timeouts through the "global stabilization time," often employed in synchronous or semisynchronous protocols, is not feasible. Consequently, akin to the work of Kursawe [12], nodes dispatch messages addressed to themselves within the network. Upon receiving such a message, the timer increments, followed by the message being sent again. This process continues until the counter maintained by the timer reaches the threshold τ, signifying a timeout.

3.2. Component Definitions

Our goal is to achieve an atomic broadcasting protocol (ABC). ABC is frequently implemented via ACS, where ACS typically comprises components such as AVID, ABA, and MVBA, which are defined as follows:

AVID (Asynchronous Verifiable Information Dispersal)

The purpose of the AVID protocol is to disperse a message from a sender to other nodes in a group of nodes, which may include adversaries. An AVID protocol has two functions: `DISPERSE` (M), which a dispersing client invokes to disperse a message M to n nodes, and `RETRIEVE`, which a (possibly different) retrieving client invokes to retrieve the message M. AVID is frequently employed for implementing RBC (reliable broadcast) and satisfies the following properties:

- Termination: If an honest client invokes DISPERSE (M) and no other client invokes DISPERSE at the same instance, then every honest node eventually finishes the dispersal phase.
- Agreement: If any honest node finishes the dispersal phase, all honest nodes eventually finish the dispersal phase.
- Availability: If an honest node has finished the dispersal phase, and some honest client initiates RETRIEVE, then the client eventually reconstructs some message M'.
- Correctness: If an honest node has finished the dispersal phase, then honest clients always reconstruct the same message M' when invoking RETRIEVE. Furthermore, if an honest client invoked DISPERSE (M) and no other client invokes DISPERSE at the same instance, then $M' = M$.

ABA (Asynchronous Binary Agreement Protocol)

The ABA protocol's purpose is for a group of nodes, which may include adversaries, to each provide a 0 or 1 as input. At the end of the protocol, each node outputs one bit and satisfies the following properties:

- Agreement: If an honest node outputs b, then all honest nodes output b.
- Termination: If all honest nodes provide the same input b, then all honest nodes output b.
- Validity: If an honest node outputs b, then at least one honest node had b as input.

MVBA (Multivalued Validated Byzantine Agreement)

MVBA allows agreement on arbitrary values instead of being restricted to binary values. The protocol has a global, polynomial-time computable predicate Q known to all nodes, which is determined by the particular application. The basic idea of the protocol is that each party proposes a (different) value that contains certain validation information as input and outputs a value that satisfies Q as the decision value. The protocol ensures that the decision value was proposed by at least one party. Each honest node only inputs a value to MVBA that satisfies Q. MVBA satisfies the following properties:

- Termination: If every honest node P_i inputs with an externally valid value v_i, then every honest node outputs a value;
- External validity: If an honest node outputs a value v, then $Q(v) = True$;
- Agreement: All honest nodes that terminate output the same value;
- Integrity: If all nodes are honest and if some nodes output v, then some nodes proposed v.

ACS (Asynchronous Common Subset Protocol)

The ACS protocol runs in a system composed of V nodes, where the number of adversaries satisfies $n \geqslant 3f + 1$. Each node proposes a set of suggested values as input. At the end of the protocol, each node selects a subset of these suggested values as output, satisfying the following properties:

- Validity: If an honest node outputs a set S of suggested values, then S contains at least $2f + 1$ suggested values, with at least $f + 1$ coming from honest nodes.
- Agreement: If an honest node outputs a set S of suggested values, then all honest nodes output S.
- Totality: If every honest node provides an input to the ACS protocol, then eventually, all honest nodes will receive an output.

4. FlexBFT Design

This section delineates the algorithmic design of FlexBFT, a pioneering optimistic asynchronous BFT protocol. Distinguishing itself from other protocols of its kind, FlexBFT introduces a unique approach to perform persistently in back-up mode, augmented by supplementary mechanisms aimed at enhancing both its efficiency and scalability.

4.1. FlexBFT Algorithm Structure (Overview)

As shown in Figure 2, FlexBFT can be broadly divided into two parts at a high level: fast mode and back-up mode. Fast mode and back-up mode are designed to address semisynchronous and asynchronous network environments, respectively. In the initial state, each node's default entry point is fast mode. Fast mode internally implements an optimistic combination of snw-ABC and pace-sync, which will be mentioned below. In fast mode, nodes can rapidly achieve consensus in semisynchronous network conditions and handle some Byzantine behavior of nodes. Nodes continuously achieve consensus rapidly in fast mode through epochs. However, when fast mode encounters an asynchronous network where no progress can be made, it transitions to back-up mode while ensuring that honest nodes have the same progress. Back-up mode is implemented with mature

asynchronous consensus protocols, ensuring liveness even in asynchronous conditions but suffering from higher latency due to the requirement for randomness. It is worth noting that after FlexBFT enters back-up mode, the consensus process will be repeated and pushed forward in back-up mode until progress can be made in fast mode before returning to fast mode. The persistence of back-up mode results in FlexBFT not wasting too many communication cycles on the fast lane rollback when facing continuous network fluctuations. In addition, FlexBFT adopts a parallelized dideterministic protocols and asynchronous protocols approach, which can not only obtain the opportunity to restart fast mode, but also effectively shorten the high latency caused by MVBA.

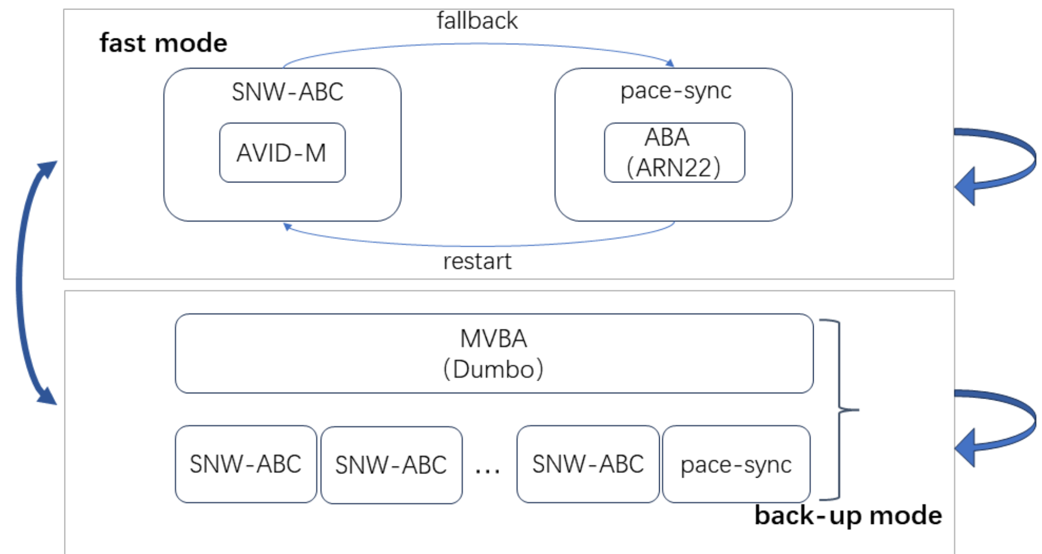

Figure 2. The total framework of FlexBFT.

4.2. Fast Mode Protocol

The fast mode of FlexBFT consists of two components: snw-ABC and pace-sync. snw-ABC and pace-sync play roles similar to HotStuff's chain-based consensus and view-change mechanisms. They enable fast consensus that can tolerate up to $N \geqslant 3f + 1$ faults and provide some support for asynchronous environments. One of the innovations in FlexBFT lies in efficiently utilizing the properties of AVID to avoid bandwidth waste caused by traditional consensus mechanisms that require downloading and agreeing on entire blocks.

The entry point of the fast mode protocol is snw-ABC, as shown in Algorithm 1. Initially, all nodes activate snw-ABC instances. As depicted in the figure, snw-ABC in this paper is implemented using AVID and threshold signatures. However, snw-ABC does not achieve consensus on an entire block; instead, it focuses on the data availability of the block. Therefore, snw-ABC will output the availability proof of the transaction, denoted as BAP (block availability proof). Following the ideas of DispersedLedger [28], snw-ABC has nodes voting on the hash value or Merkle root of the transactions rather than the block itself. Then download the block parallelly.

Utilizing AVID in snw-ABC. The key to decoupling consensus from block downloads lies in AVID, as introduced before. AVID consists of two steps: disperse and retrieve. Disperse divides a block into multiple parts using RS encoding and sends them to other replica nodes. This reliable broadcast ensures message delivery even in asynchronous environments. Each AVID instance emits a complete event, indicating the completion of dispersion. Retrieve, on the other hand, retrieves these partial data pieces from different nodes and combines them into a complete block. AVID effectively distributes the network bandwidth burden in consensus scenarios (compared with centralized broadcasting) and supports a workflow where dispersion and proof acquisition precede block assembly.

Algorithm 1 snw_ABC

// variable Declarations
1: let id be the identification of the epoch
2: let buf be a FIFO queue of input
3: let B be the batch parameter
4: let P_ℓ be the leader where $\ell = (id \mod n) + 1$
5: P_i initializes $s = 1$, $\sigma_0 = \perp$, and runs the protocol in consecutive slot number s as:
//during the Dispersal phase
6: **if** P_i is the leader P_ℓ **then**
7: activate AVID$[\langle id, s \rangle].dispersal()$ with input $TXs_s \leftarrow buf[:B]$
8: **else**
9: activate AVID$[\langle id, s \rangle].dispersal()$ as nonleader party
10: **end if**
//during the Vote phase after dispersal is completed
11: **upon** AVID$[\langle id, s \rangle]$ returns $\mathcal{H}(TXs_s)$:
12: let $\sigma_{s,i}$ be the partial signature for $\langle id, s, \mathcal{H}(TXs_s) \rangle$
13: send $Vote(id, s, \sigma_s, i)$ to P_j for each $j \in [N]$
//during the Commit phase after received enough vote
14: **upon** receiving $2f + 1$ $Vote(id, s, \sigma_{s,i})$ from distinct parties P_i:
15: let σ_s be the valid full signature that aggregates different $2f + 1$ $\sigma_{s,i}$
16: let $Proof_s := \langle H(TXs_s), \sigma_s \rangle$
17: **output** $BAP := \langle id, s, Proof_s \rangle$
18: let $s \leftarrow s + 1$
//exit condition for the algorithm
19: **upon** $abandon(id)$ is invoked:
20: abort the above execution

In the implementation, as shown in Algorithm 1, the first step of snw-ABC is to broadcast transactions proposed by the leader invoking AVID. The leader acts as an AVID server, while other nodes act as clients, initiating AVID dispersal. Simultaneously, they perform threshold signatures on $\langle id, s, \mathcal{H}(TXs) \rangle$ to cast votes, indicating that the data for TXs are available. When a node receives $2f + 1$ votes, it aggregates them into a complete signature, which, from the perspective of the partially synchronous network's consensus protocol, represents a QC (quorum certificate). snw-ABC first outputs $\mathcal{H}(TXs)$ and the aggregated threshold signature as BAP to the buffer and proceeds to download the complete transactions after the function *retrieve* is invoked. Block downloading commences in parallel immediately after the output of $\mathcal{H}(TXs)$.

The next stage of the fast mode protocol involves using ABA to complement snw-ABC, as shown in Algorithm 2, which lacks agreement properties, achieving pace synchronization. Due to snw-ABC's notarizability, it can be assumed that the consensus progress of correct nodes is mostly in the state of $(s, s-1)$ or $(s+1, s)$, where s represents the slot number. Thus, achieving agreement among correct nodes only requires agreeing on one of two consecutive natural numbers. We employ ABA to achieve consensus on the maximum slot number. For each node, ABA takes the input max(slot number)%2. Subsequently, based on the output of ABA, the current consensus progress $syncPace_e$ for correct nodes is determined.

Algorithm 2 Fast mode protocol

```
    // variable Initialization
 1: let e ← 0 be the epoch number
 2: let Proof_e ←⊥ be proof of block in epoch e
 3: let pending_e be the temporary storage area of blocks
    // consensus on block available proof
 4: activate snw_ABC[e] instance, invoke timer(τ).start()
 5: upon snw_ABC[e] delivers a BAP:
 6:     parse BAP := ⟨e, p, Proof_p⟩
 7:     invoke snw_ABC[e].retrieve(Proof_p) to download full transactions
    // commit after the current block in position
 8: upon TXs_p is delivered:
 9:     if verify(TXs_p, Proof_p) is True:
10:         output pending_e
11:         buf ← buf − {TXsinpending_e}
12:         pending_e ← block := ⟨e, p, TXs_p, Proof_p⟩
13:         p_e ← p, Proof_e ← p_e,timer.restart()
    // fall back to pace_sycn
14: upon timer expires or ∃tx ∈ buf which was buffered Tclocks ago:
15:     invoke snw_ABC.abandon()
16:     multicast FALLBACK(e, p_e,Proof_e)
17: upon receiving n − f FALLBACK(e, p_e^j,Proof_e^j):
18:     invoke pace_sync(e), invoke back-up mode protocol and wait for its return to
    continue
19:     proceed to the next epoch e ← e + 1
```

Since pace synchronization may result in a consensus progress rollback, the protocol introduces a pending buffer. The fast mode protocol remains pending as TXs ready to be committed to the log. During the pace synchronization phase, based on the return value of ABA, two cases exist:

- If $syncPace_e = p_e$, it indicates that the TXs in pending status are the latest and should be committed directly.

- If $syncPace_e \leq p_e + 1$, it means that for that node, all blocks have not been entirely retrieved. In this case, nodes must retrieve the chunk root of the p_e block, which may have already reached consensus through AVID, or even copy it from the log of correct nodes. Subsequently, they commit it to their blocks.

Malicious actions by the leader, snw-ABC timeouts, or censorship threats trigger a transition to pace synchronization. Pace synchronization can handle most of the issues in a partially synchronous Byzantine environment and aims to keep the protocol in the fast mode protocol as much as possible. However, it can still encounter fully asynchronous network problems, in which case, the fast mode protocol loses liveness. Pace synchronization identifies that snw-ABC has made no progress through ABA and then enters back-up mode until returning to fast mode.

4.3. Back-Up Mode Protocol

The back-up mode protocol is proposed to handle asynchronous network conditions and ensure liveness even in the presence of adversarial interference. Typically, back-up protocols for asynchronous networks directly use ABC as a black box. However, while ACS offers good throughput, it often suffers from significant latency due to the use of randomness-based methods. Recently, ParBFT [39] restructured the framework of optimistic asynchronous consensus, proposing a method that runs the optimistic path and pessimistic path in parallel and uses ABA for nodes to reach consensus on which path to choose. This method guarantees the effectiveness of asynchronous environments and ensures rapid output in good network conditions by treating the optimistic path as a fast

lane. Our proposed approach also aims to make the back-up mode protocol as efficient as possible through parallel processing. However, unlike ParBFT, FlexBFT does not consider the output of the optimistic path as a signal to abort the ongoing pessimistic path since, in a sense, this would be detrimental to censorship. An adversary could manipulate the network to determine whether the consensus result for that round comes from the optimistic or pessimistic path. Moreover, in cases where the output of the optimistic path is in plaintext, the adversary could delay transactions as desired.

As shown in Algorithm 3, the back-up mode protocol consists of two modules running in parallel. The first module provides the primary functionality of the back-up mode protocol, enabling it to operate successfully and produce blocks in adverse network conditions, serving as the ABC module. The second module, considering algorithm reusability, comprises the snw-ABC and pace synchronization introduced in the fast mode protocol, ensuring that even in the presence of favorable networks, it competes for block output.

Algorithm 3 Back-up mode protocol

// invoke Optimistic path
1: let $e' \leftarrow 0$ be the epoch number
2: let $Blocks_o \leftarrow [\,]$
3: activate snw_ABC[e] instance
4: **upon** snw_ABC[e]delivers a BAP:
5: activate snw_ABC[e'].retrieve$(Proof_p)$
6: **upon** TXs_o is delivered:
7: **output** $pending'_e$
8: $Blocks_o \leftarrow Block_o + pending'_e$
9: let $Block_o$ be the combination of BAP and TXs_o
10: $Blocks_o \leftarrow Block_o + pending'_e$
11: $pending'_e \leftarrow Block_o$
// invoke Pessimistic path
12: let $txs_i \leftarrow$ randomly select [B / n]-sized transactions from the top of buf
13: let $x_i \leftarrow$ threshold-encrypted txs_i
14: let $\{x_j\} \leftarrow$ dumbo$[e'](x_i)$
15: let $block_p \leftarrow$ decrypted$\{x_j\}$
// decide return fast mode protocol or not
16: **upon** $block_p \neq \emptyset$:
17: $block_p \leftarrow block_p - \{TX \text{ in } Blocks_o\}$
18: **output** $block_p$
19: **if** $Blocks_o \neq \emptyset$:
20: multicast RESTART(e', $p_{e'}$,$Proof_{'e'}$)
21: **Upon** receiving $n - f$ RESTART(e, p^j_e,$Proof^j_e$):
22: invoke pace_sync[e'], invoke fast mode protocol and wait its return to continue
23: proceed to the next epoch $e' \leftarrow e' + 1$

The construction of the ABC module has many works to refer to, but most of them build ABC based on Caching's work using ACS to construct ABC. We construct the ABC module based on Dumbo as a black box, even though Dumbo may not be the optimal ACS. Considering censorship, nodes must perform threshold encryption on transactions before inputting them into the Dumbo instance and then decrypt and output them.

Optimistic output in back-up mode. The optimistic path in the back-up mode protocol consists of a series of snw-ABC and a final pace synchronization, similar to the construction of the fast mode protocol. Their purpose is to opportunistically output TXs in a segment of good network conditions after entering back-up mode, minimizing the overall latency of back-up mode. First, continuous snw-ABC runs as normal; if pace synchronization is activated due to Byzantine faults, timeouts, etc., snw-ABC is not restarted. In another case, when the pessimistic path (ABC implemented with dumbo) is outputting, pace synchronization is activated to achieve agreement in the optimistic path, which ensures that

all correct nodes have the same output on the optimistic path. Additionally, the optimistic path records all TXs output in this round.

Persistence of back-up mode. When both modules complete running in parallel, each optimistic path has output batches of transactions $Blocks_o$, while the pessimistic path has a large batch of $Block_p$ to be output. At this point, to prevent duplicate submissions of TXs for each node, output $Block_p - \{TXinBlocks_o\}$ is implemented. If $Blocks_o$ is not empty, the next round returns to the fast mode protocol; otherwise, the back-up mode protocol continues running. Simply put, if the optimistic path akin to fast mode cannot operate and output transactions while in back-up mode, it indicates that the network conditions have not improved. In such a scenario, reverting to fast mode would be futile, as there is a high likelihood that fast mode will also be unable to make progress. It would be preferable to persist in back-up mode. Conversely, whenever the optimistic path manages to output, FlexBFT perceives this as an opportunity to return to fast mode and initiates the next epoch in fast mode to conduct a general and fast consensus process.

4.4. Performance Analysis

As an optimistic asynchronous consensus algorithm, FlexBFT needs to analyze its performance from a variety of network conditions. The main things that need to be paid attention to are communication complexity, message complexity, and block generation delay.

- When network conditions are good, nw-ABC in the fast mode protocol is mainly run. Its message complexity is $O(n^2)$, and its communication complexity is $O(nB)$. It requires four rounds of communication to output a block.
- When there are some network fluctuations, what promotes consensus is nw-ABC and pace synchronization. The ABA in pace synchronization here is implemented by the work of Ittai et al. [36], which is almost the most advanced ABA at present. Each ABA requires seven rounds of communication.
- When there are strong network fluctuations, except for the first fallback from the fast mode protocol to the back-up mode protocol, which requires all the above overhead plus the communication complexity of MVBA $O(n^2|m| + \lambda n^3 \log n)$ and $O(n^3)$ message complexity.

5. Performance Evaluation

5.1. Implementation

We implemented FlexBFT using Python. Our implementation was mainly improved from BDT's open-source code, and naturally also used some methods used by BDT, such as basic cryptography libraries and threshold signatures. In addition to some components that are not open source or written in Python, including ABA by Itta et al. [36], AVID-M is implemented in Golang. In addition, we know that Dumbo is no longer the most advanced MVBA, but we still use it as part of the back-up mode of FlexBFT. This is because our research is not optimistic about the performance of MVBA.

In addition to FlexBFT, we have implemented Dumbo, HotStuff, and BDT as comparative algorithms. Each of these represents a distinct approach in the realm of Byzantine-fault-tolerant (BFT) consensus protocols: Dumbo is a next-generation classical asynchronous BFT protocol, HotStuff is renowned for its efficiency in a semisynchronous BFT protocol, and BDT stands as a state-of-the-art (SOTA) optimistic asynchronous BFT protocol. We believe that by comparing FlexBFT with these algorithms, we can effectively highlight the distinctive features and advantages of FlexBFT.

We use Docker on a large computing server to simulate blockchain nodes, setting each node allocated 10 cores and 50 Mbps bandwidth and, similarly, operating the network delay through Docker's built-in network driver.

During the implementation process, we will fill the buffer of each node with random strings (as dummy transactions). As consensus proceeds, logs are printed and the relevant parameters TPS and latency are calculated, as shown in Equations (1) and (2), where TPS evaluates the number of transactions that a replica can successfully record per second,

and latency (default unit is milliseconds (ms)) measures the time cost from the proposed block to the replica record. In Equation (2), *recordTime* represents the time when a replica appends a new block including transactions to its shared ledger, and requestTime represents the time when a client generates a transaction and gives it to the blockchain node. All performance assessments are conducted consecutively, and their results are averaged from five separate trials.

$$TPS = \frac{success\ transactions}{totalTime} \tag{1}$$

$$Latency = recordTime - requestTime \tag{2}$$

5.2. Result

In our first set of experiments, we focused on the relationship between the number of nodes and the performance of the algorithms under favorable network conditions. We set the timeout parameter to be greater than the actual network delay in the experiments and increased the number of replica nodes from 4 to 28 with a step size of 6. The experimental results are presented in Figures 3 and 4.

The results indicate that as the number of replica nodes increases, all four algorithms exhibit significant performance scalability, each with its own emphasis on different aspects of performance. Asynchronous protocols demonstrate the ability to achieve both high throughput and low latency, even with 28 replica nodes.

Regarding latency, HotStuff, BDT, and FlexBFT can maintain approximately 300 ms of low latency even with 28 replica nodes. In the case of BDT and FlexBFT, this is primarily due to the operation of Blot and the fast mode protocol, respectively. Their low latency is attributed to the absence of randomness, resulting in lower communication complexity. However, Dumbo, constrained by the complex MVBA, still faces a message complexity of n^3, leading to poor latency performance, as expected. Furthermore, FlexBFT's latency is slightly lower than BDT, likely due to the separation of block download and consensus in the fast mode strategy. In terms of throughput, a situation similar to the previous case emerges, with Dumbo, HotStuff, and FlexBFT enjoying higher throughput, hovering around 10,000 transactions per second. The differences in throughput can be attributed to the AVID method used by these three protocols, which efficiently distributes the system's bandwidth load. In the case of HotStuff, where transactions are broadcasted, the main node's bandwidth becomes the bottleneck.

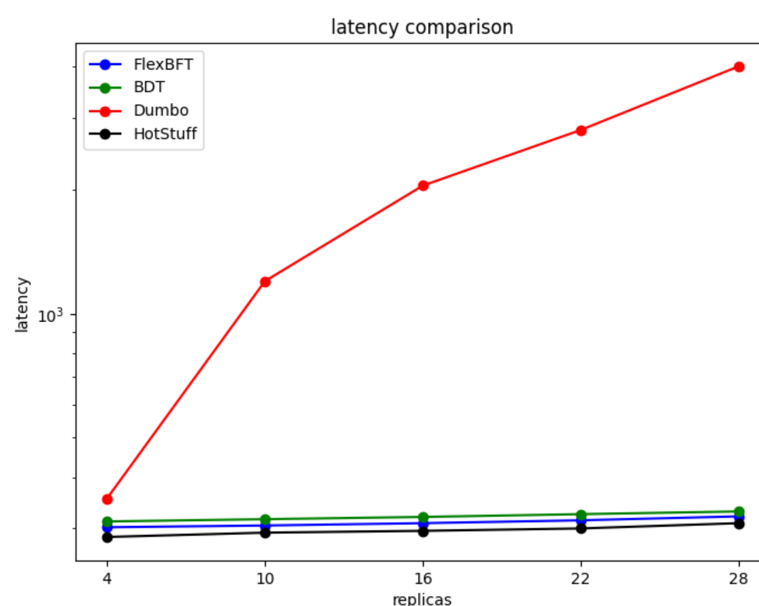

Figure 3. Latency comparison.

Figure 5 illustrates the relationship between latency and throughput under different settings (batch size). This setting introduces worse network conditions with more network fluctuations, reflecting the trade-off between latency and throughput when system bandwidth and other resources are fixed. It is worth noting that, in this scenario, a significant performance gap exists between BDT and FlexBFT. On average, for the same throughput, FlexBFT exhibits 31.6% less latency compared with BDT. This improvement is attributed to FlexBFT's strategy of back-up mode persistence, reducing the redundant fallback overhead.

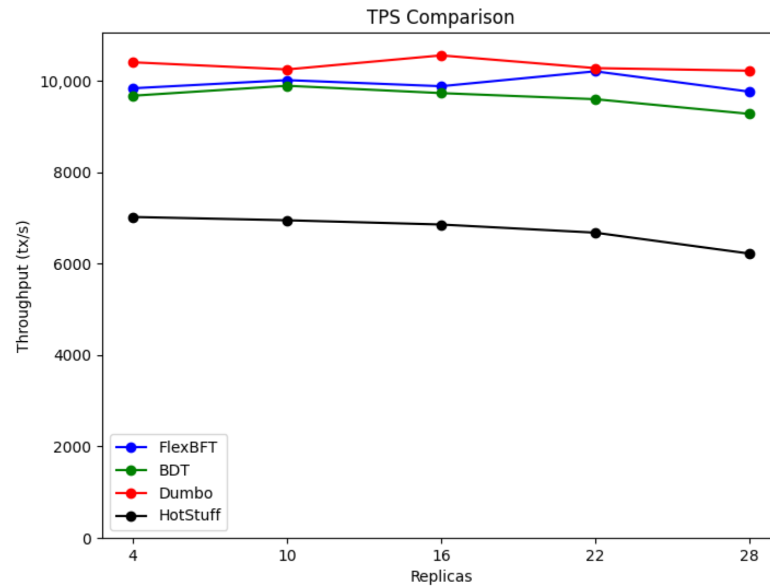

Figure 4. Throughout comparison.

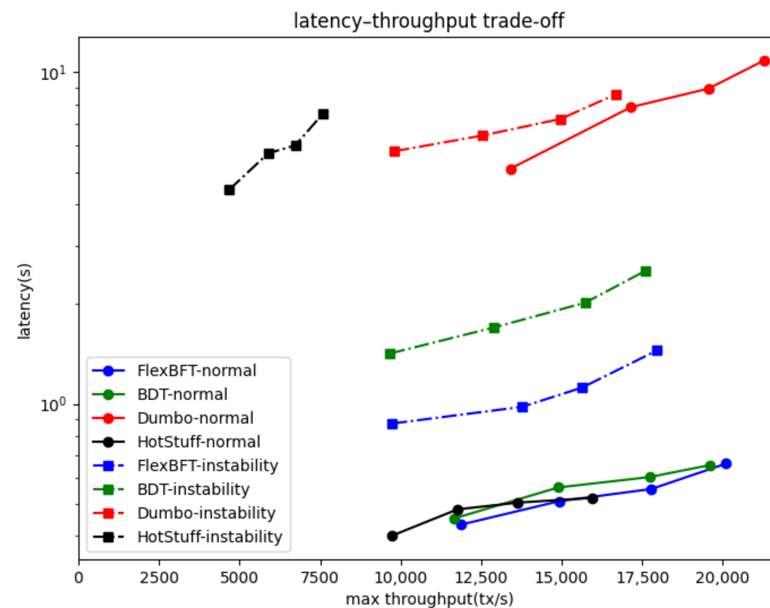

Figure 5. Comparative throughput–latency performance.

Table 1 presents an intuitive performance comparison under the same parameters (with 16 nodes and a batch size of 5000). The first two columns show data from a normal network environment, where it can be observed that FlexBFT slightly outperforms BDT. The primary comparison here is between FlexBFT's fast mode and BDT's bolt module, with the difference attributed to FlexBFT's fast mode decoupling AVID, separating consensus from block downloading, resulting in performance enhancement. The latter two columns compare

performance under worse network conditions, similar to what is illustrated in Figure 5, highlighting the superiority of persisting in back-up mode and the optimistic output within back-up mode.

Table 1. Comparison of performance of FlexBFT and BDT in the same parameter.

	Normal Network		Challenging Network	
	FlexBFT	**BDT**	**FlexBFT**	**BDT**
Latency(s)	0.309	0.320	0.980	1.425
Throughput	10232	9731	6819	5811

6. Discussion

FlexBFT is an optimistic asynchronous BFT protocol designed to balance the efficiency of synchronous and semisynchronous BFT with the liveness of asynchronous BFT. This paper compares it experimentally with the semisynchronous protocol HotStuff, the asynchronous protocol Dumbo, and the state-of-the-art (SOTA) optimistic asynchronous BFT protocol BDT, demonstrating its superior performance.

The primary contribution of FlexBFT is the concept of persisting in the slow lane. Unlike previous works that follow a linear, fixed execution path, FlexBFT can remain in the slow lane, or the back-up mode mentioned earlier, when network conditions are persistently poor. This approach eliminates the need to constantly fail in the fast lane and then switch to the slow lane through complex asynchronous protocols, such as ABA, before running the ACS protocol, thereby saving the overhead associated with fast lane timeouts and slow lane transitions.

Within this new framework of persisting in the slow lane, determining when to return to the fast lane and how to optimize performance in the slow lane poses another challenge. Thus, FlexBFT's back-up mode does not solely rely on the ACS protocol but concurrently operates an optimistic path composed of semisynchronous consensus and a pessimistic path formed through MVBA. These paths are activated simultaneously during back-up mode operation, with the optimistic path speculatively outputting to reduce overall latency and probe whether the network conditions are sufficient for fast mode to operate normally.

To further enhance performance, FlexBFT's fast mode, or fast lane, decouples semisynchronous consensus achieved through AVID into separate phases of consensus and downloading. This strategy makes fuller use of nodes' computational and bandwidth resources, thereby increasing the protocol's parallelism.

The experimental results of this paper first validate that FlexBFT, as an optimistic asynchronous BFT protocol, indeed manages to combine the advantages of both HotStuff and Dumbo across different environments, showing commendable performance in throughput, latency, and liveness. Second, the experimental results concerning the latency–throughput trade-off, along with those under conditions of increased network volatility, highlight FlexBFT's advantages over BDT.

7. Conclusions and Future Scope

While existing optimistic asynchronous BFT protocols can rapidly operate and possess the robustness of asynchronous consensus in favorable network conditions, they introduce significant latency in adverse network environments due to their fixed path order. To address this issue, we introduce FlexBFT, which primarily focuses on the persistence of back-up mode. This approach allows the protocol to refrain from hastily returning to the fast lane under adverse network conditions, thereby reducing the overhead of constantly switching from the fast to the slow lane in asynchronous network environments. In order to further enhance system efficiency, we adopt a strategy of running bandwidth-intensive and communication-intensive components in parallel, decoupling consensus and downloading in fast mode, and incorporating optimistic output in back-up mode. Our experimental results demonstrate that FlexBFT is a qualified optimistic asynchronous BFT, achieving

performance with 31.6% less latency than BDT under network fluctuations. In the future, with the excellent asynchronous consensus component framework of FlexBFT already in place, it becomes even more necessary to open the black box of consensus components for more integrated combinations.

Author Contributions: Conceptualization, A.S. and C.Z.; methodology, A.S. and C.Z.; software, C.Z.; validation, A.S. and C.Z.; formal analysis, A.S.; investigation, C.Z.; resources, A.S.; data curation, C.Z.; writing—original draft preparation, C.Z.; writing—review and editing, A.S. and C.Z.; visualization, C.Z.; supervision, A.S.; project administration, A.S.; funding acquisition, A.S. All authors have read and agreed to the published version of the manuscript..

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zheng, Z.; Xie, S.; Dai, H.N.; Chen, X.; Wang, H. Blockchain challenges and opportunities: A survey. *Int. J. Web Grid Serv.* **2018**, *14*, 352–375. [CrossRef]
2. Lamport, L.; Shostak, R.; Pease, M. The Byzantine generals problem. In *Concurrency: The Works of Leslie Lamport*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 203–226.
3. Castro, M.; Liskov, B. Practical byzantine fault tolerance. *OsDI* **1999**, *99*, 173–186.
4. Kotla, R.; Clement, A.; Wong, E.; Alvisi, L.; Dahlin, M. Zyzzyva: Speculative byzantine fault tolerance. *Commun. ACM* **2008**, *51*, 86–95. [CrossRef]
5. Gueta, G.G.; Abraham, I.; Grossman, S.; Malkhi, D.; Pinkas, B.; Reiter, M.; Seredinschi, D.A.; Tamir, O.; Tomescu, A. SBFT: A scalable and decentralized trust infrastructure. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 24–27 June 2019; pp. 568–580.
6. Yin, M.; Malkhi, D.; Reiter, M.K.; Gueta, G.G.; Abraham, I. HotStuff: BFT consensus with linearity and responsiveness. In Proceedings of the PODC '19: ACM Symposium on Principles of Distributed Computing, Torontoo, ON, Canada, 29 July–2 August 2019; pp. 347–356.
7. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proceedings of the EuroSys '18: Thirteenth EuroSys Conference 2018, Porto, Portugal, 23–26 August 2018; pp. 1–15.
8. Baudet, M.; Ching, A.; Chursin, A.; Danezis, G.; Garillot, F.; Li, Z.; Malkhi, D.; Naor, O.; Perelman, D.; Sonnino, A. State machine replication in the libra blockchain. *Libr. Assn. Tech. Rep.* **2019**, *1*, 1–41.
9. Miller, A.; Xia, Y.; Croman, K.; Shi, E.; Song, D. The honey badger of BFT protocols. In Proceedings of the CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 31–42.
10. Bahga, A.; Madisetti, V.K. Blockchain platform for industrial internet of things. *J. Softw. Eng. Appl.* **2016**, *9*, 533–546. [CrossRef]
11. Conley, J.P. *Blockchain and the Economics of Crypto-Tokens and Initial Coin Offerings*; Vanderbilt University Department of Economics Working Papers; Vanderbilt University: Nashville, TN, USA, 2017.
12. Kursawe, K.; Shoup, V. Optimistic asynchronous atomic broadcast. In Proceedings of the 32nd International Colloquim, ICALP 2005, Lisbon, Portugal, 11–15 July 2005; pp. 204–215.
13. Ramasamy, H.V.; Cachin, C. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In Proceedings of the 9th International Conference, OPODIS 2005, Pisa, Italy, 12–14 December 2005; pp. 88–102.
14. Lu, Y.; Lu, Z.; Tang, Q. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft. In Proceedings of the CCS '22: 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 2159–2173.
15. Gelashvili, R.; Kokoris-Kogias, L.; Sonnino, A.; Spiegelman, A.; Xiang, Z. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 296–315.
16. Lamport, L.; Shostak, R.; Pease, M. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* **1982**, *4*, 382–401. [CrossRef]
17. Dolev, D.; Strong, H.R. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.* **1983**, *12*, 656–666. [CrossRef]
18. Abraham, I.; Malkhi, D.; Nayak, K.; Ren, L.; Yin, M. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 106–118. [CrossRef]

19. Buchman, E. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. Master's Thesis, University of Guelph, Guelph, ON, Canada, 2016.
20. Kapitza, R.; Behl, J.; Cachin, C.; Distler, T.; Kuhnle, S.; Mohammadi, S.V.; Schröder-Preikschat, W.; Stengel, K. CheapBFT: Resource-efficient Byzantine fault tolerance. In Proceedings of the EuroSys '12: Seventh EuroSys Conference 2012, Bern, Switzerland, 10–13 April 2012; pp. 295–308.
21. Liu, J.; Li, W.; Karame, G.; Asokan, N. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Comput.* **2018**, *68*, 139–151. [CrossRef]
22. Ren, S.; Lee, C.; Kim, E.; Helal, S. Flexico: An efficient dual-mode consensus protocol for blockchain networks. *PLoS ONE* **2022**, *17*, e0277092. [CrossRef]
23. Ben-Or, M. Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols. In Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing—PODC '83, Montreal, QC, Canada, 17–19 August 1983; ACM Press: New York, NY, USA, 1983; pp. 27–30. [CrossRef]
24. Dolev, D. The Byzantine generals strike again. *J. Algorithms* **1982**, *3*, 14–30. [CrossRef]
25. Cachin, C.; Kursawe, K.; Petzold, F.; Shoup, V. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology—CRYPTO 2001*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 524–541. [CrossRef]
26. Cachin, C.; Poritz, J. Distributing trust on the Internet. In Proceedings of the 2001 International Conference on Dependable Systems and Networks, Gothenburg, Sweden, 1–4 July 2002; pp. 167–176. [CrossRef]
27. Duan, S.; Reiter, M.K.; Zhang, H. BEAT: Asynchronous BFT made practical. In Proceedings of the CCS '18: 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2028–2041.
28. Yang, L.; Park, S.J.; Alizadeh, M.; Kannan, S.; Tse, D. DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks. In Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), Renton, WA, USA, 4–6 April 2022; pp. 493–512.
29. Liu, C.; Duan, S.; Zhang, H. Epic: Efficient asynchronous bft with adaptive security. In Proceedings of the 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Valencia, Spain, 29 June–2 July 2020; pp. 437–451.
30. Guo, B.; Lu, Z.; Tang, Q.; Xu, J.; Zhang, Z. Dumbo: Faster asynchronous bft protocols. In Proceedings of the CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 9–13 November 2020; pp. 803–818.
31. Abraham, I.; Malkhi, D.; Spiegelman, A. Asymptotically optimal validated asynchronous byzantine agreement. In Proceedings of the PODC '19: ACM Symposium on Principles of Distributed Computing, Toronto, ON, Canada, 29 July–2 August 2019; pp. 337–346.
32. Zhou, Y.; Zhang, Z.; Zhang, H.; Duan, S.; Hu, B.; Wang, L.; Liu, J. Dory: Asynchronous BFT with Reduced Communication and Improved Efficiency. *Cryptol. Eprint Arch.* **2022**. Available online: https://eprint.iacr.org/2022/1709 (accessed on 6 February 2024).
33. Bracha, G. Asynchronous Byzantine agreement protocols. *Inf. Comput.* **1987**, *75*, 130–143. [CrossRef]
34. Alhaddad, N.; Das, S.; Duan, S.; Ren, L.; Varia, M.; Xiang, Z.; Zhang, H. Asynchronous Verifiable Information Dispersal with Near-Optimal Communication. *Cryptol. Eprint Arch.* **2022**. Available online: https://eprint.iacr.org/2022/775 (accessed on 6 February 2024).
35. Mostéfaoui, A.; Moumen, H.; Raynal, M. Signature-free asynchronous Byzantine consensus with t< n/3 and O (n2) messages. In Proceedings of the PODC '14: ACM Symposium on Principles of Distributed Computing, Paris, France, 15–18 July 2014; pp. 2–9.
36. Abraham, I.; Ben-David, N.; Yandamuri, S. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement. In Proceedings of the PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, 25–29 July 2022; pp. 381–391.
37. Fischer, M.; Lynch, N.; Paterson, M. Impossibility of distributed consensus with one faulty process. *J. ACM* **1985**, *32*, 374–382. [CrossRef]
38. Yurek, T.; Xiang, Z.; Xia, Y.; Miller, A. Long Live The Honey Badger: Robust Asynchronous {DPSS} and its Applications. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 5413–5430.
39. Dai, X.; Zhang, B.; Jin, H.; Ren, L. ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path. *Cryptol. Eprint Arch.* **2023**. Available online: https://eprint.iacr.org/2023/679 (accessed on 6 February 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.