Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT

Yuan Lu*
Institute of Software
Chinese Academy of Sciences

Zhenliang Lu*
School of Computer Science
The University of Sydney

Qiang Tang*
School of Computer Science
The University of Sydney

ABSTRACT

An urgent demand of deploying BFT consensus (e.g., atomic broadcast) over the Internet is raised for implementing (permissioned) blockchain services. The deterministic synchronous protocols can be simple and fast in good network conditions, but are subject to denial-of-service (or even safety vulnerability) when synchrony assumption fails. Asynchronous protocols, on the contrary, are robust against the adversarial network, but are substantially more complicated and slower for the inherent use of randomness.

Facing the issues, optimistic asynchronous atomic broadcast (Kursawe-Shoup, 2002; Ramasamy-Cachin, 2005) was proposed to improve the normal-case performance of the slow asynchronous consensus. They run a deterministic fastlane if the network condition remains good, and can fall back to a fully asynchronous protocol via a pace-synchronization mechanism (analog to view-change with asynchronous securities) if the fastlane fails. Unfortunately, existing pace-synchronization directly uses a heavy tool of asynchronous multi-valued validated Byzantine agreement (MVBA). When such fallback frequently occurs in the fluctuating wide-area network setting, the benefits of adding fastlane can be eliminated.

We present Bolt-Dumbo Transformer (BDT), a generic framework for *practical* optimistic asynchronous atomic broadcast. At the core of BDT, we set forth a new fastlane abstraction that is simple and fast, while preparing honest parties to gracefully face potential fastlane failures caused by malicious leader or bad network. This enables a highly efficient pace-synchronization to handle fallback. The resulting design reduces a cumbersome MVBA to a variant of the conceptually simplest *binary* agreement only. Besides detailed security analyses, we also give concrete instantiations of our framework and implement them. Extensive experiments demonstrate that BDT can enjoy both the low latency of deterministic protocols (e.g., 2-chain version of HotStuff) and the robustness of state-of-the-art asynchronous protocols in practice.

CCS CONCEPTS

• **Security and privacy** → Systems security; Distributed systems security; • **Computer systems organization** → Reliability.

*Authors are listed alphabetically. Yuan Lu and Zhenliang Lu contributed equally. Emails: luyuan@iscas.ac.cn, zhlu9620@uni.sydney.edu.au, qiang.tang@sydney.edu.au.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559346

KEYWORDS

Byzantine-fault tolerance, asynchronous consensus, optimsitic path

ACM Reference Format:
Yuan Lu, Zhenliang Lu, and Qiang Tang. 2022. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3559346

1 INTRODUCTION

The explosive popularity of decentralization [22, 62] creates an unprecedented demand of deploying robust Byzantine fault tolerant (BFT) consensus on the global Internet. These consensus protocols were conventionally abstracted as BFT atomic broadcast (ABC) to replicate an ever-growing linearized log of transactions among n parties [27]. Informally, ABC ensures *Safety* and *Liveness* despite that an adversary controls the communication network (e.g., delay messages) and corrupt some participating parties (e.g., $n/3$). Safety ensures all honest parties to eventually output the same log of transactions, and liveness guarantees that any transaction inputted by some honest party eventually appears in honest parties' output.

A desideratum for robust BFT in the absence of synchrony. The dynamic nature of Internet poses new fundamental challenges for implementing secure yet still highly efficient BFT consensus protocols. Traditionally, most practical BFT protocols were studied for the in-house scenarios where participating parties are geographically close and well connected. Unsurprisingly, their securities rely on some form of assumptions about the network conditions. For example, classic synchrony assumption needs all messages to deliver within a known delay, and its weaker variant called partial synchrony [39] (a.k.a. eventual synchrony) assumes that after an unknown global stabilization time (GST), all messages can be delivered synchronously. Unfortunately, these synchrony assumptions may not always hold in the wide-area network (WAN), because of fluctuating bandwidth, unreliable links, substantial delays, and even network attacks. What's worse, in an asynchronous network [10], such (partially) synchronous protocols [8, 9, 11, 12, 16, 30, 31, 47, 48, 67, 73] will grind to a halt (i.e., suffers from the inherent *loss of liveness* [40, 58]), and Bitcoin might even have a *safety* issue of potential double-spending [70] when the adversary can arbitrarily schedule message deliveries. That said, when the network is adversarial, relying on synchrony could lead to fatal vulnerabilities.

It becomes a *sine qua non* to consider robust BFT consensus that can thrive in the unstable or even adversarial Internet for mission-critical applications (e.g., financial services or cyber-physical systems). Noticeably, the class of fully asynchronous protocols [24, 38, 50, 58, 74] can ensure safety and liveness simultaneously without

any form of network synchrony, and thus become the arguably most robust candidates for implementing mission-critical applications.

Fully asynchronous BFT? Robustness with a high price! Nevertheless, the higher security assurance of asynchronous BFT consensus does not come for free: the seminal FLP "impossibility" [40] states that no *deterministic* protocol can ensure both safety and liveness in an asynchronous network. So asynchronous ABC must run *randomized* subroutines to circumvent the "impossibility", which already hints its complexity. Indeed, few asynchronous protocols have been deployed in practice during the past decades due to large complexities, until the recent HoneyBadgerBFT [58] and Dumbo-BFT [50] (and their latest improved variants [49, 74]) provide novel paths to *practical* asynchronous ABC in terms of realizing optimal linear amortized communication cost per output transaction.

Despite those recent progresses, the actual performance of state-of-the-art randomized asynchronous consensus is still far worse than the deterministic (partial) synchronous ones (e.g., HotStuff [75][1]), especially regarding the critical latency metric. For example, in the same WAN deployment environments consisting of n=64 or 100 Amazon EC2 instances across the globe, HotStuff is dozens of times faster than Dumbo [50]. Even worse, the inferior latency performance of asynchronous protocols stems from the fact that all parties generate some common randomness (e.g., "common coin" [25, 28]), and multiple repetitions are necessary to ensure the parties to coincidentally output with an overwhelming probability. Even if in one of the fastest existing asynchronous protocols such as Speeding Dumbo [49], it still costs about a dozen of rounds on average. While for their (partially) synchronous counterparts, only a very small number of rounds are required in the optimistic cases when the underlying communication network is luckily synchronous [7], e.g., 5 in the two-chain HotStuff and 3 in PBFT.

The above issues correspond to a fundamental "dilemma" lying in the design space of BFT consensus protocols suitable for the open Internet: the cutting-edge (partially) synchronous deterministic protocols can optimistically work very fast, but lack liveness guarantee in adversarial networks; on the contrary, the fully asynchronous randomized protocols are robust even in malicious networks, but suffer from poor latency performance in the normal case. Facing that, a natural question arises:

Can we design a BFT consensus achieving the best of both synchronous and asynchronous paradigms, such that it (i) is "as fast as" the state-of-the-art deterministic BFT consensus on the normal Internet with fluctuations and (ii) performs nearly same to the existing performant asynchronous BFT consensus even if in a worst-case asynchronous network?

1.1 Our contributions

We answer the aforementioned question affirmatively by presenting the first *practical* and *generic* framework Bolt-Dumbo Transformer (BDT) for optimistic asynchronous atomic broadcast. Here, optimistic asynchronous atomic broadcast [44, 54, 69] refers to an asynchronous consensus that has a deterministic fastlane that might luckily progress in the benign network environment (optimistic case) without randomized execution.

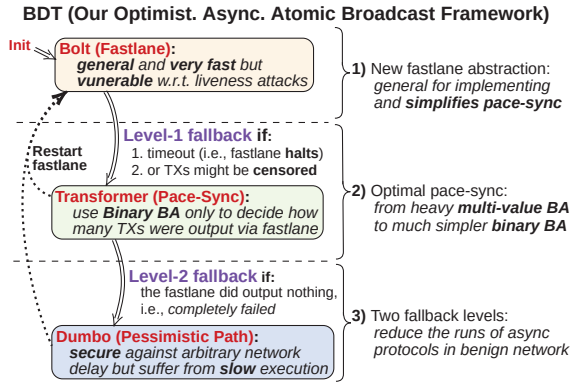

Figure 1: The overview of Bolt-Dumbo Transformer**.**

At a very high-level, BDT has three phases as shown in Fig. 1:

- *Fastlane* (nickname Bolt): It initially runs a deterministic protocol as fastlane to quickly progress in the optimistic case that synchrony assumption holds.
- *Pace-synchronization* (called pace-sync for short or nickname Transformer): If the fastlane fails to progress in time, a fast pace-synchronization mechanism (analog to view-change with asynchronous securities) is triggered to make all honest parties agree on from where and how to restart (directly restart Bolt or enter pessimistic path).
- *Pessimistic path* (nickname Dumbo [2]): In case the fastlane was completely failed to make progress, the honest parties enter a pessimistic path of asynchronous BFT to ensure liveness even in the worst case.

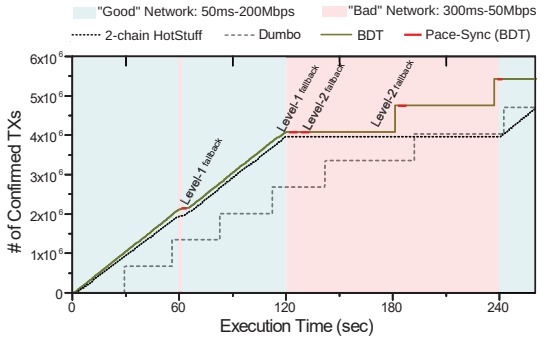

Figure 2: Simulated executions of BDT**, 2-chain** HotStuff **and** Dumbo-BFT **under fluctuating network (n=64). Bad network happens twice: one lasts 2 seconds and one lasts 120 seconds. See Section 7 for the details on simulation setup.**

BDT is featured with guaranteed *liveness* and safety even in a hostile asynchronous network with optimal tolerance against $n/3$ byzantine parties. More importantly, as depicted in Fig. 2, it realizes the best of its both paths, i.e., it is as fast as deterministic protocols (such as the state-of-the-art pipeline BFT protocols, e.g., 2-chain HotStuff) in "good" synchronous periods; and also as robust as asynchronous protocols in the "bad" network.

[1]Remark that [75] gave a 3-chain HotStuff protocol along with a 2-chain variant. Throughout the paper, we let HotStuff refer to the 2-chain version (with minor difference to fix the view-change issue) for the lower latency of the 2-chain version.

[2]Here we call the pessimistic path Dumbo and call the concrete ABC protocol in [50] Dumbo-BFT. The latter could be an instantiation for the former (as in our experiments), but other better ABC protocol could also instantiate the former pessimistic path.

Technical contributions. Different from pioneering studies [11, 47, 54, 69] that only demonstrated theoretic feasibility and had questionable practicability because of complex and slow *asynchronous* pace-synchronization (cf. Section 2 for detailed discussions on their efficiency bottleneck), BDT makes several technical contributions to harvest the best of both paths in practice. In greater detail,

A new fastlane abstraction better prepared for failures. To simplify the complicated pace-sync, we propose a new fastlane abstraction of notarizable weak atomic broadcast (nw-ABC for short) to prepare honest parties in a graceful condition when facing potential fastlane failures. Notably, nw-ABC realizes ABC in the optimistic case, and only ensures "notarizability" otherwise: any output block is with a quorum proof to attest that sufficient honest parties have received a previous block (along with valid proof). Such an nw-ABC can be easily constructed to be very fast, e.g., from a sequence of simple (provable) multicasts; and more importantly, the notarizability (as we will carefully analyze) guarantees that any two honest parties will be at neighboring blocks when entering pace-sync, so we can leverage simpler binary agreement to replace the cumbersome full-fledged asynchronous atomic broadcast or multi-value agreement used in prior art [11, 47, 54, 69].

Cheapest possible pace-synchronization. More importantly, with the preparation of nw-ABC, Transformer reduces pace-sync to a problem that we call *two-consecutive-valued Byzantine agreement* (tcv-BA), which is essentially an asynchronous *binary* Byzantine agreement (ABBA). In contrast, prior art [54, 69] leveraged cumbersome multi-valued agreement (MVBA) for pace-sync (cf. Sec. 2 for a careful review). Transformer thus improves the communication complexity of pace-sync by an $O(n)$ factor, and is essentially optimal for pace-sync, because the pace-sync problem can be viewed as a version of asynchronous consensus, and ABBA is the arguably simplest asynchronous consensus. In practice, Transformer attains a minimal overhead similar to the fastlane latency. Further care is needed for invoking nw-ABC to ensure the safety (see next section).

Avoiding pessimistic path whenever we can. To further exploit the benefits brought by fast Transformer, we add a simple check after pace-sync to create two-level fallbacks: if pace-sync reveals that the fastlane still made some output, it immediately restarts another fastlane without running the actual pessimistic path. This is in contrast with previous works [54, 69] where the slow pessimistic path will always run after each pace-sync, which is often unnecessarily costly if there are only short-term network fluctuations. Remark that the earlier studies cannot effectively adopt our two-level fallback tactic, because their heavy pace-sync might bring extra cost and it may even nullify the advantages of the fastlane in case of frequent fallbacks.

Generic framework enabling flexible instantiations. BDT is generic, as it enables flexible choices of the underlying building blocks for all three phases. For example, we present two exemplary fastlane instantiations, resulting in two BDT implementations that favor latency and throughput, respectively, so one can instantiate BDT according to the actual application scenarios. Also, Transformer can be constructed around any asynchronous binary agreement, thus having the potential of using any more efficient ABBA to further reduce the fallback overhead (e.g., the recent ABBAs from Crain [34], Das et al. [37], Zhang et al. [76] and Abraham et al. [2]). Similarly, though currently we use Dumbo-BFT as the pessimistic path, this can be replaced by more efficient recent designs [49, 74].

Extensive evaluations. To demonstrate the practical performance of BDT, we implement the framework using Dumbo-BFT [50] as the exemplary pessimistic path. We compare two typical BDT implementations to Dumbo-BFT and HotStuff, and conduct extensive experiments in real-world/simulated environments.

We first deploy all protocols in the same real-world WAN environment consisting of up to 100 Amazon EC2 c5.large instances across the globe. Some highlighting experimental results could be found in Table 1, which are: (i) the BDT implementation based on sequential multicasts can attain a basic latency about only 0.44 second (i.e., nearly same to 2-chain HotStuff's and less than 3% of Dumbo-BFT's latency), even if we intentionally raise frequent pace synchronizations after every 50 optimistic blocks; (ii) in the worst case that we intentionally make fastlane to always fail, the throughput of BDT remains 90% of Dumbo-BFT's and is close to 20,000 transaction per second, even if we do let BDT wait as long as 2.5 seconds to timeout before invoking pace-sync. Those demonstrate BDT can be "as fast as" the deterministic protocols in normal cases and can maintain robust performance in the worst case.

To understand more scenarios in-between, we then conduct evaluations in a controlled test environment that can simulate fluctuating network (i.e., switching between "good" and "bad" networks). In the simulated good network (i.e., 50 ms packet delay and 200 Mbps peer-to-peer link), BDT is almost as fast as 2-chain HotStuff; while in the simulated bad network (i.e., 300 ms packet delay and 50 Mbps peer-to-peer link), BDT can closely track the performance of underlying pessimistic asynchronous protocol, though HotStuff might grind to a halt due to inappropriately chosen timeout parameter.

See Section 7 for more detailed experiment setup and results.

Table 1: BDT, 2-chain HotStuff and Dumbo-BFT running over 100 EC2 c5.large servers across 16 regions in 5 continents

	Good-Case v.s. HotStuff (fastlanes always complete)			Worst-Case v.s. Dumbo (fastlanes always fail)	
	BDT-sCAST**	BDT-sRBC†	HotStuff§	BDT-Timeout‡	DumboBFT
Basic latency (sec)	0.44	0.67	0.42	21.95	16.36
Throughput (tx/sec)*	9,253	18,234	10,805	18,806	21,242

* Each transaction has 250 bytes to approximate the basic Bitcoin transaction.
** BDT-sCAST is BDT with using Bolt-sCAST, where Bolt-sCAST is the fastlane instantiation built from pipelined multicasts.
† BDT-sRBC is BDT with using Bolt-sRBC, where Bolt-sRBC is the fastlane instantiation built from sequential reliable broadcasts.
‡ BDT-Timeout idles for 2.5 sec in the fastlane, and then runs Pace-Sync+Dumbo.
§ Reasons why our evaluations for HotStuff different from [75]: (i) we tested among 16 AWS regions instead of one AWS region; (ii) we let the implementation to agree on 250-byte tx instead of 32-byte tx hash; (iii) we use a single-process network layer written in Python, differing from multi-processing network layer in [75].

2 EFFICIENCY BOTTLENECK OF PRIOR ART AND OUR SOLUTION IN A NUTSHELL

Efficiency obstacles in prior art. As briefly mentioned, pioneering works of Kursawe-Shoup [54] (KS02) and a following improvement of Ramasamy-Cachin [69] (RC05) initiated the study of *optimistic asynchronous atomic broadcast* by adding a deterministic fastlane to fully asynchronous atomic broadcast, and they adopted multi-valued *validated* Byzantine agreement (MVBA) to facilitate fallback once the fastlane fails to progress.

Nevertheless, these prior studies are theoretical in asynchronous networks, as they rely on heavy MVBA or even heavier full-fledged state-machine replication for fallback. Serious efficiency hurdle remains in such cumbersome fallback, thus failing to harvest the best of both paths in practice. Let us first overview the remaining hurdles and design challenges.

Challenge and effiency bottleneck lying in pace-synchronization. The fastlane of KS02 and RC05 directly employs a sequence of some broadcast primitives (the output of which is also called a block for brevity). If a party does not receive a block within a period (defined by a timeout parameter), then it requests fallback by informing other parties about the index of the block that it just received. When the honest parties receive a sufficient number of fallback requests (e.g., $2f + 1$ in the presence of f faulty parties), they execute the *pace-synchronization* mechanism to decide where to continue the pessimistic path.

Since different honest parties may have different progress in the fastlane when they decide to fall back, e.g., some are now at block 5, some at block 10, thus pace-synchronization needs to ensure: (i) all honest parties can eventually enter the pessimistic path from the same block; and (ii) all the "mess-ups" (e.g., missing blocks) left by the fastlane can be properly handled. Both requirements should be satisfied in an asynchronous network! These requirements hint that all the parties may need to *agree* on a block index that is proposed by some honest party, otherwise they might decide to sync up to some blocks that were never delivered. Unfortunately, directly implementing such a functionality requires one-shot asynchronous (multi-valued) Byzantine agreement with strong validity (that means the output must be from some honest party), which is infeasible because of inherent exponential communication [41].

Both KS02 and RC05 smartly implement pace-synchronization through asynchronous multi-valued *validated* Byzantine agreement (MVBA) to get around the infeasible strong validity. An MVBA is a weaker and implementable form of asynchronous multi-valued Byzantine agreement, the output of which is allowed to be from a malicious party but has to satisfy a predefined predicate. Still, MVBA is a cumbersome building block (and can even construct full-fledged asynchronous atomic broadcast directly [24]). What's worse, KS02 and RC05 invoke this heavy primitive for both pace-synchronization and pessimistic path, causing at least $O(n^3)$-bit communication and dozens of rounds. Although we may reduce the $O(n^3)$ communication to $O(n^2)$ by some very recent results (e.g., Dumbo-MVBA [57]), however, they remain costly in practice due to a large number of extra execution rounds and additional computing costs (e.g., erasure encoding/decoding).

Slow pace-sync remains in a more general framework [11]. Later, Aublin et al. [11] studied a more general framework that is flexible to assemble optimistic fastlanes and full-fledged BFT protocols, as long as the underlying modules all satisfy a defined Abstract functionality. To facilitate fallback when the fastlane fails due to network asynchrony or corruptions, [11] used a stronger version of Abstract variant with guaranteed liveness (called Backup). Backup can guarantee all parties to output exact k common transactions [11], so it can handle fallback by first finishing pace-sync, then deciding some output transactions (i.e., running as the pessimistic path), and finally restarting the fastlane.

Aublin et al. [11] also pointed out that Backup (with guaranteed progress) can be obtained from full-fledged BFT protocols. For example, [11] gave exemplary Backup instantiations based on PBFT [29] and Aardvark [32] in the partially synchronous setting. This indicated another feasible way to implement asynchronous fallback, i.e., implement Backup by full-fledged asynchronous BFT protocols.

Unfortunately, when Backup is implemented via full-fledged asynchronous BFT, it would be as heavy as MVBA (or even heavier), since most existing performant asynchronous BFT protocols are either constructed from MVBA [49, 50] or have implicit MVBA [44]. That said, though the framework presented in [11] is more general than KS02 and RC05, it is not better than KS02 and RC05 with respect to the efficiency of pace-sync (and thus has the same efficiency bottleneck lying in pace-sync).

In contrast, we identify an extra simple property (not covered by Abstract [11]), so the new fastlane abstraction (1) enables us to utilize a much simpler asynchronous pace-synchronization, and (2) still can be easily obtained with highly efficient instantiations.

Consequences of slow pace-synchronization. The inefficient pace-sync severely harms the practical effectiveness of adding fastlane. In particular, when the network may fluctuate as in the real-world Internet, the pace-sync phase might be triggered frequently, and its high cost might eliminate the benefits of adding fastlane.

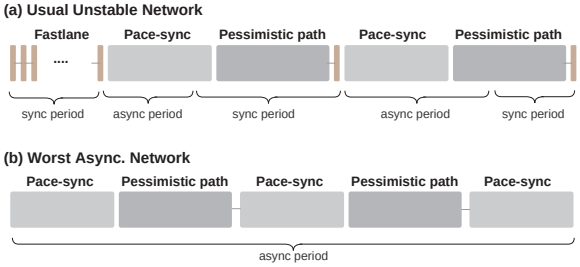

Figure 3: Consequence of slow fallback in KS02/RC05 in fluctuating networks. The length of each phase denotes latency.

To see the issue, consider the heavy pace-sync of existing work that is as slow as the asynchronous pessimistic path and dozens of times slower than the fastlane.[3] As Fig. 3 (a) exemplifies, although the network stays in good conditions for the majority of time, the overall average latency of the protocol is still way larger than its fastlane. One slow fallback could "waste" the gain of dozens of optimistic blocks, and it essentially renders the optimistic fastlane ineffective. In the extreme case shown in Fig. 3 (b), the fallback is always triggered because the fastlane leaders are facing adaptive denial-of-service attack, it even doubles the cost of simply running the pessimistic asynchronous protocol alone.

It follows that in the wide-area Internet, *inefficient pace synchronization* in previous theoretical protocols likely eliminates the potential benefits of optimistic fastlane, and thus their applicability is limited. So a fundamental practical challenge remains to minimize the overhead of pace-sync, such that we can harvest the best of both paths in optimistic asynchronous atomic broadcast.

Our Solution in a Nutshell. Now we walk through how we overcome the above challenge and reduce the complex pace-sync problem to only a variant of asynchronous binary agreement.

[3] Actual situation might be much worse in RC02 [54] because several more MVBA invocations with much larger inputs are executed in the pace-sync.

First ingredient: a new abstraction of the fastlane. We put forth a new simple fastlane abstraction called *notarizable weak atomic broadcast* (nw-ABC, with nickname Bolt). In the optimistic case, it performs as a full-fledged atomic broadcast protocol and can output a block per τ clock ticks. But if the synchrony assumption fails to hold, it won't have liveness nor exact agreement, only a *notarizability* property can be ensured: whenever any party outputs a block at position j with a valid quorum proof, at least $f+1$ honest parties already output at the position $j-1$, cf. Fig. 4.

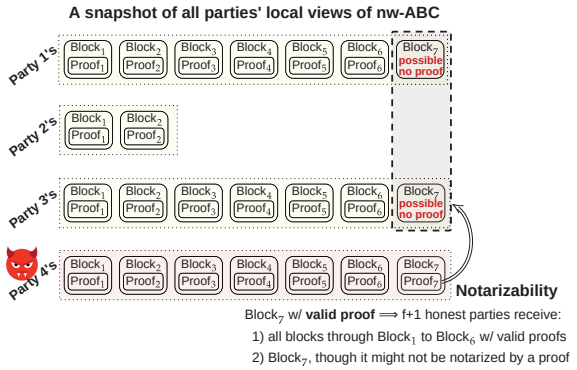

Figure 4: Notarizability of fastlane abstraction (nw-ABC).

How "notarizability" better prepares honest parties? To see how notarizability simplifies pace-sync, let us examine the pattern of the honest parties' fastlane outputs before entering pace-sync.

Suppose all honest parties have quit the fastlane, exchanged their fallback requests (containing their latest block index and the corresponding quorum proof), received such $2f+1$ fallback requests, and thus entered the pace synchronization. At the time, let s to be the largest index of all fastlane blocks with valid proofs.

We can make two easy claims: (i) no honest party can see a valid fallback request with an index equal or larger than $s+1$; (ii) all honest parties must see some fallback request with an index equal or larger than $s-1$. If (i) does not hold, following notarizability, at least one party can produce a proof for block $s+1$, which contradicts the definition of s. While for (ii), since block s is with a valid proof, at least $f+1$ honest parties received block $s-1$ with valid proof. So for any party waits for $2f+1$ fallback requests, it must see at least one fallback sent from some of these $f+1$ honest parties, thus seeing $s-1$; otherwise, there would be $3f+2$ parties.

The above two claims narrow the range of the honest parties' fallback positions to $\{s-1, s\}$, i.e., *two unknown consecutive integers*.

Second ingredient: async. agreement for consecutive values. Pace-sync now is reduced to pick one value of two unknown consecutive integers $\{s-1, s\}$. To handle the problem, we further define *two-consecutive-valued Byzantine agreement* (tcv-BA), which can be easily implemented from any asynchronous *binary* Byzantine agreement (cf. Section 5 for the concrete construction).

Final piece of the puzzle: adding "safe-buffer" to the fastlane. When tcv-BA outputs u, all honest parties can sync up to block u accordingly. Because no matter u is s or $s-1$, the u-th fastlane block is with a valid quorum proof, so it can be retrieved due to notarizability (cf. Fig. 4). Nevertheless, a subtle issue remains: tcv-BA cannot guarantee $u=s$, and thus u could be $s-1$. This is because an asynchronous adversary can always delay the messages of the honest parties that input s to make them seemingly crashed. That means, if a party outputs a fastlane block immediately when seeing its proof, it faces a threat that the pace-sync returns a smaller index and revokes the latest fastlane block. This can even temporarily violate safety requirement, if other parties output a different block after pace-sync. To solve the issue, we introduce a "safe buffer" to let the newest fastlane block be *pending*, i.e., not output it until one more fastlane block is received with a valid proof.

3 OTHER RELATED WORK

In the past decades, asynchronous BFT protocols are mostly theoretical results [3, 13–15, 21, 28, 33, 64, 65, 68], until several recent progresses such as HoneyBadgerBFT [58], BEAT [38], Dumbo-BFT protocols [49, 50, 57], VABA [6], DAG-based asynchronous protocols [35, 52], and DispersedLedger [74]. Nevertheless, they still have a latency much larger than that of good-case partially synchronous protocols. Besides the earlier discussed optimistic asynchronous consensus [54, 69] and more general framework [11], Spiegelman recently [72] used VABA [6] to instantiate pace-sync in optimistic asynchronous atomic broadcast. However, it is still inefficient, especially when fallbacks frequently occur. BDT framework presents a generic and efficient solution to add a deterministic fastlane to most existing asynchronous consensus protocols (except the DAG-based protocols). For example, it is compatible with two very recent results of DispersedLedger [74] and Speeding-Dumbo [49], and can directly employ them to instantiate more efficient pessimistic path.

It is well known that partially synchronous protocols [30, 75] can be responsive after GST in the absence of failures. Nonetheless, if some parties are slow or even act maliciously, they might suffer from a worst-case latency related to the upper bound of network delay. Some recent studies [5, 7, 48, 59, 67, 71] also consider synchronous protocols with *optimistic responsiveness*, such that when some special conditions were satisfied, they can confirm transactions very quickly (with preserving optimal $n/2$ tolerance). Our protocol is *responsive* all the time, because it does not wait for timeout that is set as large as the upper bound of network delay in all cases.

Besides, some literature [17–19, 55, 60] studied how to combine synchronous and asynchronous protocols for stronger and/or flexible security guarantees in varying network environment. We instead aim to harvest efficiency from the deterministic protocols.

Concurrent works. A concurrent work [45] considers adding an asynchronous view-change to a variant of HotStuff. Very recently its extended version [44] was presented with implementations. They focus on a specific construction of asynchronous fallback tailored for HotStuff by opening up a recent MVBA protocol [6], thus can have different efficiency trade-offs. On the other hand, they cannot inherit the recent progress of asynchronous BFT protocols to preserve the linear per transaction communication (as we do) in the pessimistic path, or future improvements (since BDT is generic). Moreover, [44] essentially still uses an MVBA to handle pace-sync, while we reduce the task to conceptual minimum—a binary agreement, which itself could have more efficient constructions.

4 PROBLEM FORMULATION

Transaction. Without loss of generality, we let a transaction denoted by tx to represent a string of $|m|$ bits.

Block structure. A block is a tuple in form of block := ⟨*epoch*, *slot*, TXs, Proof⟩, where *epoch* and *slot* are natural numbers, TXs is a sequence of transactions also known as the payload. Throughout the paper, we assume |TXs| = B, where B be the batch size parameter. The batch size can be chosen to saturate the network's available bandwidth in practice. Proof is a quorum proof attesting that at least $f + 1$ honest parties indeed vote the block by signing it.

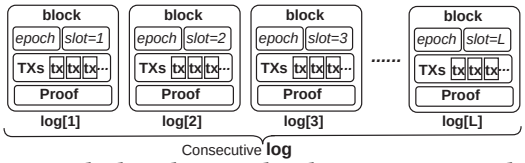

Figure 5: Block and output log due to our terminology.

Blocks as output log. Throughout the paper, a log (or interchangeably called as blocks) refers to an indexed sequence of blocks. For log with length $L := |\log|$, we might use hereunder notations. (1) $\log[i]$ denotes the i-th block in log. For example, log[1] is the first block of log, log[−1] is the alias of the last block in log, and log[−2] represents the second-to-last block in log, and so forth. (2) $\log.append(\cdot)$ can append some block to log. For example, when $\log.append(\cdot)$ takes a block $\neq \emptyset$ as input, |log| increases by one and log[−1] becomes this newly appended block; when $\log.append(\cdot)$ takes a sequence of non-empty blocks $[\text{block}_{x+1}, \ldots, \text{block}_{x+k}]$ as input, |log| would increase by k, and $\log[-1] = \text{block}_{x+k}$, $\log[-2] = \text{block}_{x+k-1}$ and so on after the operation; when $\log.append(\cdot)$ takes an empty block $\emptyset$ as input, the *append* operation does nothing.

Consecutive output log. An output log consisting of L blocks is said to be consecutive if it satisfies: for any two successive blocks $\log[i]$ and $\log[i+1]$ included by log, either of the following two cases is satisfied: (i) $\log[i].epoch = \log[i+1].epoch$ and $\log[i].slot + 1 = \log[i+1].slot$; or (ii) $\log[i].epoch + 1 = \log[i+1].epoch$ and $\log[i+1].slot = 1$. Without loss of generality, we let all logs to be *consecutive* throughout the paper for presentation simplicity.

4.1 Modeling the system and threats

We consider the standard asynchronous message-passing system with trusted setup, which can be detailed as follows.

Known identities and trusted setup. There are n designated parties, each of which has a unique identity (i.e., $\mathcal{P}_1$ through $\mathcal{P}_n$) known by everyone else. All involved threshold cryptosystems are properly set up, so all parties can get and only get their own secret keys in addition to relevant public keys. The setup can be done by a trusted dealer or distributed key generation [46, 51, 66], and the latter can be feasibly implemented in an asynchronous network as well [4, 36, 37, 43, 53].

Byzantine corruptions. The adversary can choose up to f parties to fully control before the protocol starts. Our instantiations focus on static corruptions, which is same to all recent *practical* asynchronous atomic broadcast [38, 49, 50, 58, 74]. Also, no asynchronous BFT can tolerate more than $f = \lfloor (n-1)/3 \rfloor$ Byzantine corruptions. Through the paper, we stick with this optimal resilience.

Fully-meshed reliable asynchronous network. There exists a reliable asynchronous peer-to-peer channel between any two parties. The adversary can arbitrarily delay or reorder messages, but cannot drop or modify messages sent among honest parties.

Computationally-bounded adversary. We consider computationally bounded adversary that can perform some probabilistic computing steps bounded by polynomials in the number of message bits generated by honest parties, which is standard cryptographic practice in the asynchronous network.

Adversary-controlling local "time". It is impossible to implement global time in the asynchronous model. Nevertheless, we do not require any global wall-clock for securities. Same to [23, 54], it is still feasible to let each party keep an adversary-controlling local "clock" that elapses at the speed of the actual network delay δ: each party sends a "tick" message to itself via the adversary-controlling network, then whenever receiving a "tick", it increases its local "time" by one and resends a new "tick" to itself via the adversary. Using the adversary-controlling "clock", each party can maintain a timeout mechanism, for example, let $timer(\tau).start()$ to denote that a local timer is initialized and will "expire" after τ clock ticks, and let $timer(\tau).restart()$ denote to reset the timer.

4.2 Security goal: async. atomic broadcast

Our primary goal is to develop an asynchronous atomic broadcast protocol defined as follows to attain high robustness against unstable or even hostile network environment.

Definition 4.1. In **atomic broadcast** (ABC), each party is with an implicit queue of input transactions (i.e., the input backlog) and outputs a log of blocks. Besides the syntax, the ABC protocol shall satisfy the following properties with all but negligible probability:

- *Total-order.* If an honest party outputs a log, and another honest party outputs another log′, then $\log[i] = \log'[i]$ for every i that $1 \le i \le \min\{|\log|, |\log'|\}$.
- *Agreement.* If an honest party adds a block to its log, all honest parties would eventually add the block to their logs.
- *Liveness* (adapted from [24]). If all honest parties input a transaction tx, tx would output within some asynchronous rounds (bounded by polynomials in security parameters).

Remarks on the definition of ABC. Throughout the paper, we let safety refer to the union of total-order and agreement. Besides, we insist on the liveness notion from [24] to ensure that each input transaction can output reasonably quickly instead of eventually. This reasonable aim can separate some studies that have *exponentially* large confirmation latency [13]. Moreover, the protocol must terminate in polynomial number of rounds to restrict the computing steps of adversary in the computationally-secure model [6, 24, 63], otherwise cryptographic primitives are potentially insecure.

4.3 Performance metrics and preliminaries

We are particularly interested in *practical* asynchronous protocols, and therefore, consider the following critical efficiency metrics:

- *Communication complexity.* We primarily focus on the (average) bits of all messages associated to output each block. Because the communicated bits per block essentially reflects the (amortized) communication per delivered transaction, in particular when each block includes $O(B)$-sized transactions, where B is a specified batch-size parameter.
- *Message complexity.* This characterizes the number of messages exchanged among honest parties to produce a block.

- *Asynchronous round complexity.* The eventual delivery in asynchronous network causes the protocol execution independent to "real time". Nevertheless, it is still needed to characterize the running time, and a standard way to do so is counting asynchronous "rounds" as in [24, 28].

Cryptographic abstractions. $\mathcal{H}$ denotes a collision-resistant hash function. TSIG and TPKE denote threshold signature and threshold encryption, respectively. Established TSIG is a tuple of algorithms $(\mathsf{SignShare}_t, \mathsf{VrfyShare}_t, \mathsf{Combine}_t, \mathsf{Vrfy}_t)$, and throughout the paper, we call the signature share output from $\mathsf{SignShare}_t$ the *partial signature*, and call the output of $\mathsf{Combine}_t$ the *full signature*. Established TPKE consists three algorithms $(\mathsf{Enc}_t, \mathsf{DecShare}_t, \mathsf{Dec}_t)$. In all notations, the subscript t represents the threshold, cf. some classic literature such as [58]. The cryptographic security parameter is denoted by λ, capturing the bit-length of signatures and hashes.

Building blocks. We might use the following asynchronous broadcast/consensus protocols in the black-box manner.

Definition 4.2. **Reliable broadcast (**RBC**)** has a designated sender who aims to send its input to all parties, and satisfies the next properties except with negligible probability: (i) *Validity*. If the sender is honest and inputs v, then all honest parties output v; (ii) *Agreement*. The outputs of any two honest parties are same; (iii) *Totality*. If an honest party outputs v, then all honest parties output v.

Definition 4.3. **Asynchronous binary Byzantine agreement (**ABBA**)** [25, 28, 61] has a syntax that each party inputs and outputs a single bit b, where b ranges over $\{0, 1\}$, and shall guarantee the following properties except with negligible probability: (i) *Validity*. If any honest party outputs b, then at least one honest party takes b as input; (ii) *Agreement*. The outputs of any two honest parties are same; (iii) *Termination*. If all honest parties activate the protocol with taking a bit as input, then all honest parties would output a bit in the protocol.

Definition 4.4. **Asynchronous common subset (**ACS**)** [15] has a syntax that each honest party inputs a value and outputs a set of values, and there are n participating parties with up to f corrupted parties. It satisfies the next properties except with negligible probability: (i) *Validity*. The output set S of an honest party contains the inputs of at least $n - 2f$ honest parties; (ii) *Agreement*. The outputs of any two honest parties are same; (iii) *Termination*. If all honest parties activate the protocol, then all honest parties would output.

5 ALLSPARK: FASTLANE ABSTRACTION AND TWO-CONSECUTIVE-VALUE BA

The simple and efficient pace-synchronization is the crux of making BDT practical, and this becomes possible for two critical ingredients, i.e., a novel fastlane abstraction (nw-ABC) and a new variant of binary Byzantine agreement (tcv-BA). Specifically,

- nw-ABC ensures that all parties' fastlane outputs are somewhat weakly consistent, namely, if the (s)-th block is the latest block with valid quorum proof, then at least $f + 1$ honest parties must already output the $(s - 1)$-th block with the valid quorum proof (cf. Fig. 4).
- Considering the above property of nw-ABC fastlane, we can conclude that: after exchanging timeout requests, all honest parties either know s or $s - 1$. We thus lift the conventional binary agreement to a special variant (tcv-BA) for deciding a common value out of $\{s - 1, s\}$, despite that the adversary might input arbitrarily, say $s - 2$ or $s + 1$.

5.1 Abstracting and constructing the fastlane

We first put forth notarizable weak atomic broadcast (nw-ABC) to better prepare the fastlane for more efficient pace-sync.

Definition 5.1. **Notarizable weak atomic broadcast** (nw-ABC, nicknamed by Bolt). In the protocol with an identification id, each party takes a transaction buffer as input and outputs a log of blocks, where each block $\mathsf{log}[j]$ is in form of $\langle \mathsf{id}, j, \mathsf{TXs}_j, \mathsf{Proof}_j \rangle$. There also exists two external functions Bolt.verify and Bolt.extract taking id, slot j and Proof_j as input (whose outputs and functionalities would soon be explained below). We require that Bolt satisfies the following properties except with negligible probability:

- *Total-order.* Same to atomic broadcast.
- *Notarizability.* If any (probably malicious) party outputs $\mathsf{log}[j] := \langle \mathsf{id}, j, \mathsf{TXs}_j, \mathsf{Proof}_j \rangle$ s.t. $\mathsf{Bolt.verify}(\mathsf{id}, j, \mathsf{Proof}_j) = 1$, then: there exist at least $f + 1$ honest parties, each of which either already outputs $\mathsf{log}[j]$, or already outputs $\mathsf{log}[j - 1]$ and can invoke Bolt.extract function with valid Proof_j to extract $\mathsf{log}[j]$ from received protocol scripts.
- *Abandonability.* An honest party will not output any block in Bolt[id] after invoking abandon(id). In addition, if $f + 1$ honest parties invoke abandon(id) before output $\mathsf{log}[j]$, then no party can output valid $\mathsf{log}[j + 1]$.
- *Optimistic liveness.* There exist a non-empty collection of optimistic conditions to specify the honesty of certain parties, s.t. once an honest party outputs $\mathsf{log}[j]$, it will output $\mathsf{log}[j + 1]$ in κ asynchronous rounds, where κ is a constant.

Comparing to ABC, nw-ABC does not have the exact agreement and liveness properties: (i) notarizability compensates the lack of agreement, as it ensures that whenever a party outputs a block $\mathsf{log}[j]$ at position j, at least $f + 1$ honest parties already output at the position $j - 1$, and in addition, $f + 1$ honest parties already receive the protocol scripts carrying the payload of $\mathsf{log}[j]$, so they can extract the block $\mathsf{log}[j]$ once seeing valid Proof_j; (ii) liveness is in an optimistic form, which enables simple deterministic implementations of nw-ABC in the asynchronous setting.

Careful readers might notice that the above fastlane abstraction, in particular the notarizability property, share similarities with the popular lock-commit paradigm widely used in (partially) synchronous byzantine/crash fault tolerant protocols [9, 30, 39, 48, 75]. For example, when any honest party outputs some value (i.e. "commit"), then at least $f + 1$ honest parties shall receive and already vote this output (i.e. "lock"). In such a sense, the fastlane can be easily instantiated in many ways through the lens of (partially) synchronous protocols. Unsurprisingly, one candidate is the fastlane used in KS05 [69]. Here we present two more exemplary Bolt constructions.

Comparing with the Abstract *component in [11].* As aforementioned, [11] defined Abstract as a basic component to compose full-fledged BFT consensus with optimistic fastlane. Abstract was defined to capture a very broad array of optimistic conditions including very optimistic cases such as no fault at all, such that a fastlane satisfying Abstract definition could be designed as simple as possible (with the price that no guarantee of progress among honest parties in the

presence of faults, as nw-ABC can, before triggering fallback). For example, [11] presented Quorum, an implementation of Abstract that only involves one round trip with an optimistic condition allowing no fault, but Quorum cannot meet the critical notarizability property of nw-ABC. Taking Quorum as example, the weakening of Abstract prevents us from using binary agreement to handle the failed Abstract fastlanes, because the parties cannot reduce the failed position of the fastlane to two consecutive numbers. This corresponds to the necessity of our stronger nw-ABC definition in the context of facilitating a simplest possible pace-sync in the asynchronous setting.

Bolt **from sequential multicasts**. As shown in Fig. 6, Bolt can be easily constructed from pipelined multicasts with using threshold signature, and we call it Bolt-sCAST. The idea is as simple as: the leader proposes a batch of transactions via multicast, then all parties send back their signatures on the proposed batch as their votes, once the leader collects enough votes from distinct parties (i.e., $2f+1$), it uses the votes to form a quorum proof for its precedent proposal, and then repeats to multicast a new proposal of transactions (along with the proof). Upon receiving the new proposal and the precedent proof, the parties output the precedent proposal and the proof (as a block), and then vote on the new proposal. Such execution is repeated until the abandon interface is invoked.

let id be the session identification of Bolt[id], buf be a FIFO queue of input, B be the batch parameter, and $\mathcal{P}_\ell$ be the leader (where $\ell = (\text{id} \mod n) + 1$)

$\mathcal{P}_i$ initializes $s = 1$, $\sigma_0 = \perp$ and runs the protocol in consecutive slot number s as:

- **Broadcast**. if $\mathcal{P}_i$ is the leader $\mathcal{P}_\ell$
  - if $s > 1$ then:
    - wait for $2f+1$ Vote(id, $s-1$, $\sigma_{s-1,i}$) from distinct parties $\mathcal{P}_i$, where $\sigma_{s-1,i}$ is the valid partial signature signed by $\mathcal{P}_i$ for $\langle \text{id}, s-1, \mathcal{H}(\text{TXs}_{s-1})\rangle$
    - compute σ_{s-1}, the full-signature for $\langle \text{id}, s-1, \mathcal{H}(\text{TXs}_{s-1})\rangle$, by aggregating the $2f+1$ received valid partial signatures
  - multicast Proposal(id, s, TXs_s, σ_{s-1}), where $\text{TXs}_s \leftarrow$ buf[: B]
- **Commit and Vote**. upon receiving Proposal(id, s, TXs_s, σ_{s-1}) from $\mathcal{P}_\ell$
  - if $s > 1$ then:
    - proceed only if σ_{s-1} is valid full signature that aggregates $2f+1$ partial signatures for $\langle \text{id}, s-1, \mathcal{H}(\text{TXs}_{s-1})\rangle$, otherwise abort
    - output block:=(id, $s-1$, TXs_{s-1}, Proof_{s-1}), where $\text{Proof}_{s-1} := \langle \mathcal{H}(\text{TXs}_{s-1}), \sigma_{s-1}\rangle$
  - send Vote(id, s, $\sigma_{s,i}$) to the leader $\mathcal{P}_\ell$, where $\sigma_{s,i}$ is the partial signature for $\langle \text{id}, s, \mathcal{H}(\text{TXs}_s)\rangle$, then let $s \leftarrow s+1$
- **Abandon**. upon abandon(id) is invoked then: abort the above execution

Figure 6: Bolt **from sequential multicasts** (Bolt-sCAST)**. The external functions are presented in Fig. 8.**

Bolt **from sequential reliable broadcast**. As shown in Fig. 7, we can also use sequential RBC instances to implement Bolt. In the implementation, a designated fastlane leader can reliably broadcast its proposed transaction batches one by one. For each party receives a batch from some RBC, it signs the batch and RBC's identifier, and multicasts the signature as vote, then wait for $2f+1$ valid votes to form a quorum proof, such that the batch and the proof assemble an output block, and the party proceeds into the next RBC. Note that a RBC implementation [58] can use the technique of verifiable information dispersal [26] for communication efficiency as well as balancing network workload, such that the leader's bandwidth usage is at the same order of other parties'. In contrast, Bolt-sCAST might cause the leader's bandwidth usage n times more than the other parties', unless an additional mempool layer is implemented to further decouple the dissemination of transactions from Bolt.

let id be the session identification of Bolt[id], buf be a FIFO queue of input, B be the batch parameter, and $\mathcal{P}_\ell$ be the leader (where $\ell = (\text{id} \mod n) + 1$)

$\mathcal{P}_i$ initializes $s = 1$, $\sigma_0 = \perp$ and runs the protocol in consecutive slot number s as:

- **Broadcast**. if $\mathcal{P}_i$ is the leader $\mathcal{P}_\ell$, activates RBC[$\langle \text{id}, s\rangle$] with input $\text{TXs}_s \leftarrow$ buf[: B]; else activates RBC[$\langle \text{id}, s\rangle$] as non-leader party
- **Vote**. upon RBC[$\langle \text{id}, s\rangle$] returns TXs_s
  - send Vote(id, s, $\sigma_{s,i}$) to all, where $\sigma_{s,i}$ is the partial signature for $\langle \text{id}, s, \mathcal{H}(\text{TXs}_s)\rangle$
- **Commit**. upon receiving $2f+1$ Vote(id, s, $\sigma_{s,i}$) from distinct parties $\mathcal{P}_i$, where $\sigma_{s,i}$ is the valid partial-signature signed by $\mathcal{P}_i$ for $\langle \text{id}, s, \mathcal{H}(\text{TXs}_s)\rangle$
  - output block:=(id, s, TXs_s, Proof_s), where $\text{Proof}_s := \langle \mathcal{H}(\text{TXs}_s), \sigma_s\rangle$ and σ_s is the valid full signature that aggregates the $2f+1$ partial signatures for $\langle \text{id}, s, \mathcal{H}(\text{TXs}_s)\rangle$, then let $s \leftarrow s+1$
- **Abandon**. upon abandon(id) is invoked then: abort the above execution

Figure 7: Bolt **from sequential** RBCs (Bolt-sRBC)**. The external functions are presented in Fig. 8.**

// Validate Proof_s to check whether at least $f+1$ honest parties output the s-th block or can extract it (according to the next Bolt.extract function)
external function Bolt.verify(id, s, Proof_s):
  parse Proof_s as $\langle h_s, \sigma_s\rangle$
  return $\text{TSIG.Vrfy}_{2f+1}(\langle \text{id}, s, h_s\rangle, \sigma_s)$

// Leverage the valid Proof_s to extract the s-th block from some received protocol messages (though the block was not output yet).
external function Bolt.extract(id, s, Proof_s):
  if Bolt.verify(id, s, Proof_s) = 1, then parse Proof_s as $\langle h_s, \sigma_s\rangle$
  if TXs_s was received during executing Bolt s.t. $h_s = \mathcal{H}(\text{TXs}_s)$, then:
    return block:=(id, s, TXs_s, Proof_s), where $\text{Proof}_s := \langle h_s, \sigma_s\rangle$
  return block:=(id, s, $\perp$, $\perp$)

Figure 8: Invocable external functions for Bolt **instantiations**

Analysis of the Bolt **constructions**. The security analyses of Bolt-sCAST and Bolt-sRBC are simple by nature, and the complexities can be easily counted as well. The details of these analyses can be found in our online full version [56].

5.2 Two-consecutive-value BA

Another critical ingredient is a variant of binary agreement that can help the honest parties to choose one common integer out of two unknown but consecutive numbers. Essentially, tcv-BA extends the conventional binary agreement and can be formalized as follows.

Definition 5.2. **Two-consecutive-value Byzantine agreement** (tcv-BA) satisfies termination, agreement and validity (same to those of asynchronous binary agreement) with overwhelming probability, if all honest parties input a value in $\{v, v+1\}$ where $v \in \mathbb{N}$.

For each party $\mathcal{P}_i$, make the following modifications to the ABBA code in Alg. 7 of [50] (originally from [61] but with some adaptions to use Ethan MacBrough's suggestion [1] to fix the potential liveness issues of [61]):

Replace line 13-23 of Algorithm 7 in [50] with the next instructions:

- $c \leftarrow \text{Coin}_r.\text{GetCoin}()$
  - if $S_r = \{v\}$ then:
    - if $v\%2 = c\%2$
      - if $decided$ = false then: output v; $decided$ = true
      - else (i.e, $decided$ = true) then: halt
    - $\text{est}_{r+1} \leftarrow v$
  - if $S_r = \{v_1, v_2\}$ then:
    - if $v_1\%2 = c\%2$, then $\text{est}_{r+1} \leftarrow v_1$
    - else (i.e, $v_2\%2 = c\%2$), then $\text{est}_{r+1} \leftarrow v_2$

Figure 9: tcv-BA **protocol. Lines different to Alg. 7 in [50] are in orange texts.**

To squeeze extreme performance of pace synchronization, we give a non-black-box construction tcv-BA that only has to revise three lines of code of the practical ABBA construction adapted from [61]. This non-black-box construction basically reuses the protocol

pseudocode except several if-else checking (see Fig. 9) and hence has the same performance of this widely adopted ABBA protocol.

In addition, tcv-BA can be constructed from any ABBA with only 1-2 more "multicast" rounds, cf. Figure 10. This black-box construction provides us a convenient way to inherit any potential improvements of the underlying ABBA primitive [2, 34, 37, 76].

Let ABBA be any asynchronous binary agreement, then party $\mathcal{P}_i$ executes:

Upon receiving input R then:

- multicast VALUE(id, R)
- upon receiving VALUE(id, R') from $f+1$ parties containing the same R'
  - if VALUE(id, R') has not been sent before, then: multicast VALUE(id, R')
- wait for receiving $2f+1$ VALUE(id, v) messages from distinct parties carrying the same v, and activate ABBA[id] with $v\%2$ as input
- wait for ABBA[id] returns b:
  - if $v\%2 = b$, then: return v
  - else: wait for receiving $f+1$ VALUE(id, v') messages from distinct parties containing the same v' such that $v'\%2 = b$, then return v'

Figure 10: tcv-BA **protocol built from any** ABBA **"black-box"**

6 Bolt-Dumbo Transformer FRAMEWORK

As Fig. 11 outlines, the fastlane of BDT is a Bolt instance wrapped by a timer. If honest parties can receive a new Bolt block in time, they would restart the timer to wait for the next Bolt block. Otherwise, the timer expires, and the honest parties multicast a fallback request containing the latest Bolt block's quorum proof that they can see.

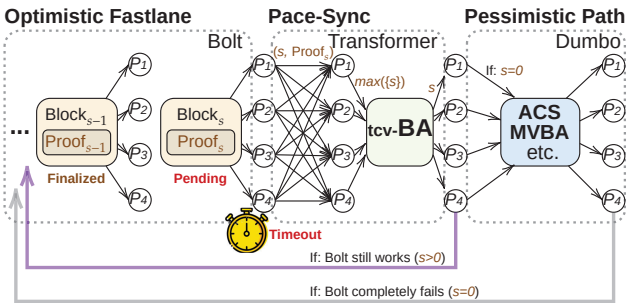

Figure 11: The execution flow of Bolt-Dumbo Transformer

After timeout, each party waits for $n-f$ fallback requests with valid Bolt block proofs, and enters pace-sync. They invoke tcv-BA with using the maximum block index (slot) in the received fallback requests as input. Eventually, the honest parties enter the tcv-BA and decide to either retry the fastlane or start the pessimistic path. As we briefly mentioned before, the reason we can use such a simple version of *binary* agreement is that via a careful analysis, we can find that nw-ABC prepares all honest parties will enter tcv-BA with taking a number out of two neighboring indices as input.

The remaining non-triviality is that tcv-BA cannot ensure its output to always be the larger number out of the two possible inputs, that means the globally latest Bolt block can be revoked after pace-sync. Hence, the latest fastlane block is marked as "pending", and a "pending" block is finally output until the fastlane returns another new block. This pending fastlane block ensures safety in BDT.

Protocol details. BDT is formally illustrated in Fig. 12. It employs a reduction to nw-ABC, tcv-BA, and some asynchronous consensus (e.g., ACS). Informally, it proceeds as follows by successive epochs:

(1) Bolt *phase.* When an honest party enters an epoch e, it activates a Bolt[e] instance, and locally starts an adversary-controlling "timer" that expires after τ clock "ticks" and resets once hearing the "heartbeat" of Bolt[e] (e.g., Bolt[e] returns a new block). If one party receives a new Bolt[e] block in time without "timeout", it temporarily records the block as pending, finalizes the previous (non-empty) pending block as BDT's output, and sets its "pace" p_e to the new block's slot number. Otherwise, the "timeout" mechanism interrupts, and the party abandons Bolt[e]. Beside the above "timeout" mechanism to ensure Bolt progress in time, we also consider that some transactions are probably censored: if the oldest transaction (at the top of the input backlog) is not output for a duration T, an interruption is also raised to abandon Bolt[e]. Once a party abandons Bolt[e] for any above reason, it immediately multicasts latest "pace" p_e with the corresponding block's proof via a PACESYNC message.

(2) Transformer *phase.* If an honest party receives $(n-f)$ valid PACESYNC messages from distinct parties w.r.t. Bolt[e], it enters Transformer. In the phase, the party chooses the maximum "pace" maxPace out of the $n-f$ "paces" sent from distinct parties, and it would use this maxPace as input to invoke the tcv-BA[e] instance. When tcv-BA[e] returns a value syncPace, all parties agree to continue from the syncPace-th block in tcv-BA[e]. In some worse case that a party did not yet receive all blocks up to syncPace, it can fetch the missing blocks from other parties by calling the CallHelp function (cf. Fig. 13).

(3) Pessimistic *phase.* This phase may not be executed unless the optimistic fastlane of the current epoch e makes no progress at all, i.e, syncPace = 0. In the worst case, Pessimistic is invoked to guarantee that some blocks (e.g., one) can be generated despite an adversarial network or corrupt leaders, which becomes the last line of defense to ensure the critical liveness.

Help **and** CallHelp**.** Besides the above main protocol procedures, a party might invoke the CallHelp function to broadcast a CALLHELP message, when it realizes that some fastlane blocks are missing. As Fig. 13 illustrates, CALLHELP messages specify which blocks to retrieve, and every party also runs a Help daemon to handle CALLHELP messages. Actually, any honest party that invokes CallHelp can eventually retrieve the missing blocks, because at least $f+1$ honest parties indeed output the blocks under request. The Help daemon can also use the techniques of erasure-code and Merkle commitment tree in verifiable information dispersal [26, 58], such that it only responds with a coded fragment of the requested blocks, thus saving the overall communication cost by an $O(n)$ order.

Alternative pessimistic path. The exemplary Pessimistic path invokes Dumbo to output one single block. Nonetheless, this is not the only design choice. First, BDT is a generic framework, and thus it is compatible with many recent asynchronous BFT protocols such as DispersedLedger [74] and not restricted to Dumbo-BFT. Second, there could be some global heuristics to estimate how many blocks needed to generate during the pessimistic path according to some public information (e.g., how many times the fastlane completely fails in a stream). Designing such heuristics to better fit real-world Internet environments could be an interesting engineering question to explore in the future but does not impact any security analysis.

Security intuitions. We brief security intuitions in the following and defer detailed proofs to the full version [56] due to space limit. *Safety*. The core ideas of proving agreement and total-order are:

- Transformer *returns a common index*. All honest parties must obtain the same block index from Transformer, so they always agree the same fastlane block to continue the pessimistic path (or retry the fastlane). This is ensured by tcv-BA's agreement.
- Transformer *returns an index not "too large"*. For the index returned from Transformer, at least $f+1$ honest parties did receive all blocks (with valid proofs) up to this index. As such, if any party misses some blocks, it can easily fetch the correct blocks from these $f+1$ parties. This is because the notarizability of Bolt prevents the adversary from forging a proof for a fastlane block with an index higher than the actually delivered block. So no honest party would input some index of an irretrievable block to tcv-BA, and then the validity of tcv-BA simply guarantees the claim.
- Transformer *returns an index not "too small"*. No honest party would revoke any fastlane block that was already committed as a finalized output. Since each honest party waits for $2f+1$ PaceSync messages from distinct parties, then due to the notarizability of Bolt, there is at least one PaceSync message contains $s-1$, where s is the latest fastlane block (among all parties). So every honest party at least inputs tcv-BA with $s-1$. The validity of tcv-BA then ensures the output at least to be

// **Optimistic Path** (also the BDT protocol's main entry)
Every party $\mathcal{P}_i$ runs the protocol in consecutive epoch numbered e (initialized as 1) as follows:

- initialize: $p_e \leftarrow 0$, $\text{Proof}_e \leftarrow \perp$, $\text{Paces}_e \leftarrow \{\}$, $\text{pending}_e \leftarrow \emptyset$
- activate Bolt[e] instance, and start a timer that expires if not being restarted after τ clock "ticks" (i.e., invoke $timer(\tau).start()$)
- upon Bolt[e] delivers a block:
  - parse block:=$\langle e, p, \text{TXs}_p, \text{Proof}_p \rangle$, where p is the "slot" number
  - $\log.append(\text{pending}_e)$, $\text{buf} \leftarrow \text{buf} \setminus \{\text{TXs in pending}_e\}$, $\text{pending}_e \leftarrow$ block //finalized the elder pending block, pending the newly fastlane block
  - $p_e \leftarrow p$, $\text{Proof}_e \leftarrow \text{Proof}_p$, $timer(\tau).restart$ // Bolt[e] makes progress in time, so restart the "heartbeat" timer
- upon $timer(\tau)$ expires or the front tx in the backlog buf was buffered T clock "ticks" ago:
  - invoke Bolt[e].abandon() and multicast PaceSync(e, p_e, Proof_e) // the fastlane is probably stucking or censoring certain transactions
- upon receiving message PaceSync(e, p_e^j, Proof_e^j) from $\mathcal{P}_j$ for the first time:
  - if Bolt.verify(e, p_e^j, Proof_e^j) = 1: $\text{Paces}_e \leftarrow \text{Paces}_e \cup p_e^j$
  - if $|\text{Paces}_e| = n - f$: // enough parties have already quitted the fastlane
    - invoke **Transformer(e)** and wait for its return to continue // enter into the pace-synchronization phase
    - proceed to the next epoch $e \leftarrow e+1$ // restart the fastlane of next epoch

// **Pace Synchronization**
internal function Transformer(e): // **internal** function shares all internal states of the BDT protocol

- let $\text{maxPace}_e \leftarrow \max(\text{Paces}_e)$ and then $\text{syncPace}_e \leftarrow$ tcv-BA[e](maxPace_e) // see Fig. 9 or 10 for concrete implementations of tcv-BA
- if $\text{syncPace}_e > 0$:
  - send PaceSync(e, syncPace_e, Proof) to all if $\text{syncPace}_e \in \text{Paces}_e$, where Bolt.verify($e$, syncPace_e, Proof) = 1
  - if $\text{syncPace}_e = p_e$: $\log.append(\text{pending}_e)$ and $\text{buf} = \text{buf} \setminus \{\text{TXs in pending}_e\}$
  - if $\text{syncPace}_e = p_e + 1$:
    - wait for a valid PaceSync(e, syncPace_e, Proof), then $\text{block}' \leftarrow$ Bolt.extract(e, syncPace_e, Proof) // try to extract the missing block
    - if block′ is in form of (e, syncPace_e, $\perp$, $\perp$), then: $\text{block}' \leftarrow$ CallHelp(e, p_e, 1) // failed to extract, have to rely on other parties to fetch, cf. Fig. 13
    - $\log.append(\text{pending}_e).append(\text{block}')$ and $\text{buf} = \text{buf} \setminus \{\text{TXs in pending}_e \text{ and block}'\}$
  - if $\text{syncPace}_e > p_e + 1$: $\text{blocks} \leftarrow \text{CallHelp}(e, p_e, \text{syncPace}_e - p_e)$ // contact other parties to fetch missing fastlane blocks, cf. Fig. 13
    - $\log.append(\text{pending}_e).append(\text{blocks})$ and $\text{buf} = \text{buf} \setminus \{\text{TXs in pending}_e \text{ and blocks}\}$
  - continue **Optimistic Path** with $e \leftarrow e+1$
- if $\text{syncPace}_e = 0$: invoke **Pessimistic(e)** and wait for its return, then continue **Optimistic Path** with $e \leftarrow e+1$

// **Pessimistic Path**
internal function Pessimistic(e): // **internal** function shares all internal states of the BDT protocol

- $\text{txs}_i \leftarrow$ randomly select $\lfloor B/n \rfloor$-sized transactions from the first B-sized transactions at the top of buf
- $x_i \leftarrow \text{TPKE.Enc}(epk, \text{txs}_i)$, namely, encrypt txs_i to obtain x_i
- $\{x_j\}_{j \in S} \leftarrow \text{ACS}[e](x_i)$, where $S \subset [n]$ and $|S| \geq n - f$
- For each $j \in S$, jointly decrypt the ciphertext x_j to obtain txs_j, so the payload $\text{TXs} = \bigcup_{j \in S} \text{txs}_j \leftarrow \{\text{TPKE.Dec}(epk, x_j)\}_{j \in S}$
- let block := $\langle e, 1, \text{TXs}, \perp \rangle$, then $\log.append(\text{block})$ and $\text{buf} = \text{buf} \setminus \{\text{TXs in block}\}$

Figure 12: The Bolt-Dumbo Transformer **(BDT) protocol**

// **The** Help **daemon process**
Help: It is a daemon process that can read the finalized output log of Bolt-Dumbo Transformer, and it listens to the down below event:

- upon receiving message CallHelp(e, tip, gap) from party $\mathcal{P}_j$ for the first time:
  - assert $1 \leq \text{gap} \leq Esize$
  - wait for log containing the block := $\langle e, \text{tip} + \text{gap}, *, * \rangle$
  - let $M \leftarrow$ retrieve all blocks in log from block := $\langle e, \text{tip}+1, *, * \rangle$ to block := $\langle e, \text{tip} + \text{gap}, *, * \rangle$
    - let $\{m_k\}_{k \in [n]}$ be the fragements of a $(n-2f, n)$-erasure code applied to M and h be a Merkle tree root computed over $\{m_k\}_{k \in [n]}$
    - send Help(e, tip, gap, h, m_i, b_i) to $\mathcal{P}_j$ where m_i is the i-th erasure-code fragement of M and b_i is the i-th Merkle tree branch

// **The** CallHelp **function**
external function CallHelp(e, tip, gap):

- let $F \leftarrow [\,]$ to be a dictionary structure such that $F[h]$ can store all leaves committed to Merkle tree root h
- multicast message CallHelp(e, tip, gap)
- upon receiving the message Help(e, tip, gap, h, m_j, b_j) from party $\mathcal{P}_j$ for the first time:
  - if b_j is a valid Merkle branch for root h and leaf m_j then: $F[h] \leftarrow F[h] \cup (j, m_j)$; otherwise discard the message
  - if $|F[h]| = n - 2f$ then:
    - interpolate the $n-2f$ leaves stored in $F[h]$ to reconstruct M, then parse M as a sequence of blocks and return blocks

Figure 13: Help **and** CallHelp**.** Help **is a daemon process having access to the output** log**, and** CallHelp **is a function to call** Help

$s-1$ as well. Recall that there is a "safe buffer" to hold the latest fastlane block as a pending one, the claim is then correct.

- *Pessimistic path and fastlane are safe.* Pessimistic path is trivially safe due to its agreement and total order. Fastlane has total-order by definition, and its weaker agreement (notarizability) is complemented by Transformer as argued above.

Liveness. This stems from the liveness of all three phases. The liveness of fastlane is guaranteed by the "timeout" parameter τ. That means, all honest parties can leave the fastlanes without "getting stuck". After that, all parties would invoke tcv-BA and obtain syncPace as the tcv-BA output due to the termination of tcv-BA; moreover, if any honest party realizes that it misses some fastlane blocks after obtaining syncPace, it can sync up to syncPace within only two asynchronous rounds, because at least $f+1$ honest parties can help it to fetch the missing blocks. So no honest party would "stuck" during the Transformer phase. Finally, the honest parties would enter the Pessimistic phase if the fastlanes completely fail to output nothing. After that, the protocol must output expected $O(B)$-sized transactions, and ensures that any transactions (at the B-top of all honest parties' backlogs) can output with a constant probability, thus ensuring liveness even if in the worst case.

Efficiency analysis. The complexities can be analyzed by counting these of each underlying module. Overall, BDT would cost expected amortized $O(n)$ bits per output transaction (for sufficiently large batch size), and the latency of each output block is of expected constant rounds. The complexities hold in all cases despite network conditions, cf. our full version [56] for detailed complexity analysis.

Optimistic conditions. BDT has a simple and efficient deterministic fastlane that might keep on progressing under certain optimistic conditions, which intuitively are: (i) the actual network delay is smaller than some guessed timing parameter and (ii) the leader of fastlane is honest. Recall that these optimistic conditions are same/comparable to the preconditions of liveness in many partially BFT consensuses such as PBFT and HotStuff.

7 PERFORMANCE EVALUATION

Implementation details. We implement BDT, Dumbo and HotStuff in the same language (i.e. Python 3), with using the same libraries and security parameters for all cryptographic implementations. The BFT protocols are implemented by single-process code. Besides, a common network layer is programmed by using unauthenticated TCP sockets. The network layer is implemented as a separate Python process to provide non-blocking communication interface.

For common coin, it is realized by hashing Boldyreva's pairing-based unique threshold signature [20] implemented over MNT224 curve. For quorum proofs, we concatenate ECDSA signatures that are implemented over secp256k1 curve. For threshold public key encryption, the hybrid encryption approach implemented in HoneyBadger BFT is used [58]. For erasure coding, the Reed-Solomon implementation in the zfec library is adopted. For timeout mechanism, we use the clock in each EC2 instance to implement the local time in lieu of the adversary-controlling "clock" in our formal security model. Our code used in evaluations is available at https://github.com/yylluu/BDT.

For notations, BDT-sCAST denotes BDT using Bolt-sCAST as fastlane, while BDT-sRBC denotes the other instantiation using Bolt-*s*RBC. In addition, BDT-Timeout denotes to use an idle fastlane that just waits for timeout, which can be used as benchmark to "mimic" the worst case that the fastlanes always output nothing due to constant denial-of-service attacks.

We present the most intuitive experiments in the section, and more evaluations can be found in our online full version [56].

7.1 Evaluations in wide-area network

Setup on Amazon EC2. To demonstrate the practicability of BDT in realistic wide-area network (WAN), we evaluate it among Amazon EC2 c5.large instances (2 vCPUs and 4 GB RAM) for n=64 and 100 parties, and also test Dumbo and HotStuff in the same setting as reference points. All EC2 instances are evenly distributed in 16 AWS regions, i.e., Virginia, Ohio, California, Oregon, Central Canada, São Paulo, Frankfurt, Ireland, London, Paris, Stockholm, Mubai, Seoul, Singapore, Tokyo and Sydney. All evaluation results in the WAN setting are measured back-to-back and averaged over two executions (each run for 5-10 minutes).

In the WAN setting tests, we might fix some parameters of BDT to intentionally amplify the fallback cost. For example, let each fastlane interrupt after output only 50 blocks, so Transformer is frequently invoked. We also set the fastlane's timeout parameter τ as large as 2.5 sec (nearly twenty times of the one-way network latency in our test environment), so all fallbacks triggered by timeout would incur a 2.5-second overhead in addition to the Transformer's latency.

Basic latency. We firstly measure the basic latency to reflect how fast the protocols are (in the good cases without faults or timeouts), if all blocks have nearly zero payload (cf. Fig. 14). This provides us the baseline understanding about how fast BDT, HotStuff and Dumbo can be to handle the scenarios favoring low-latency.

When $n = 100$, BDT-sCAST is 36x faster than Dumbo, and BDT-sRBC is 23x faster than Dumbo; when $n = 64$, BDT-sCAST is 18x faster than Dumbo, and BDT-sRBC is 10x faster than Dumbo; moreover, the execution speed of both BDT-sCAST and BDT-sRBC are at the same magnitude of HotStuff. In particular, the basic latency of BDT-sCAST is almost as same as that of 2-chain Hotstuff, which is because the fastlane of BDT-sCAST can be thought of a stable-leader 2-chain Hotstuff and its optimistic latency has five rounds [4], i.e., same to that of 2-chain Hotstuff.

Peak throughput. We then measure throughput in unit of transactions per second (where each transaction is a 250 bytes string to approximate the size of a typical Bitcoin transaction). The peak throughput is depicted in Fig. 15, and gives us an insight how well BDT, HotStuff and Dumbo can handle transaction burst. [5]

BDT-sCAST realizes a peak throughput about 85% of HotStuff's when either n is 100 or 64, BDT-sRBC achieves a peak throughput that is as high as around 90% of Dumbo's for $n = 64$ case and about 85% of Dumbo's for $n = 100$ case. All these throughput numbers are achieved despite frequent Transformer occurrence, as we intend to let each fastlane to fallback after output mere 50 blocks.

[4] The five-round latency of BDT-sCAST in the best cases can be counted as follows: one round for the leader to multicast the proposed batch of transactions, one round for the parties to vote (by signing), one round for the leader to multicast the quorum proof (and thus all parties can get a pending block), and finally two more rounds for every parties to receive one more block and therefore output the earlier pending block. The concrete of rounds of BDT-sRBC in the best cases can be counted similarly.

[5] Note that if we further implement an additional mempool as in [42] and [35], we can expect much higher throughout (that always closely tracks available bandwidth).

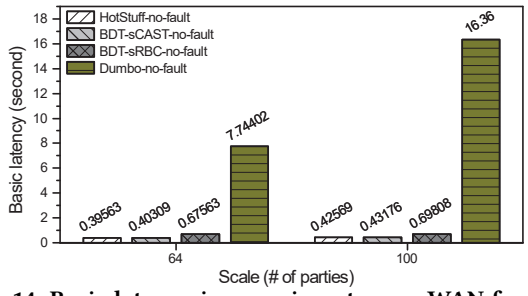

Figure 14: Basic latency in experiments over WAN for two-chain HotStuff**,** BDT-sCAST**,** BDT-sRBC **and** Dumbo**.**

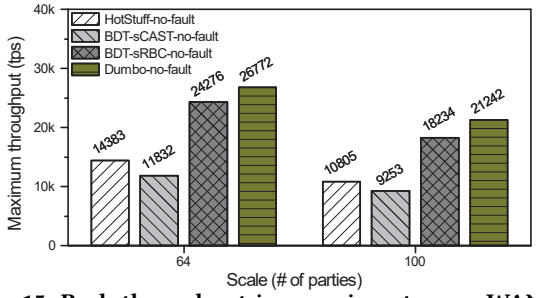

Figure 15: Peak throughput in experiments over WAN for two-chain HotStuff**,** BDT-sCAST**,** BDT-sRBC **and** Dumbo**.**

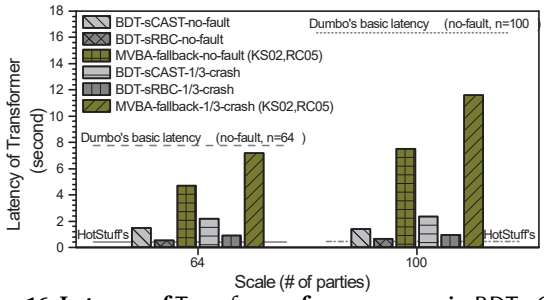

Figure 16: Latency of Transformer **for pace-sync in** BDT-sCAST **and** BDT-sRBC **(when no fault and 1/3 crash, respectively). MVBA fallback in RC05 is also tested as a reference point.**

Overhead of Transformer. It is critical for us to understand the practical cost of Transformer. We estimate such overhead from two different perspectives as shown in Fig. 16 and 17.

As shown in Fig. 16, we measure the execution time of Transformer in various settings by taking combinations of the following setups: (i) BDT-sCAST or BDT-sRBC; (ii) on or off of 1/3 crashes; (iii) 64 or 100 EC2 instances. Moreover, in order to comprehensively compare Transformer with the prior art [11, 54, 69], we also measure the latency of MVBA pace-sync (which instantiates the Backup/Abstract primitive in [11] to combine the fastlane and Dumbo [6]) as a basic reference point, cf. Section 2 for the idea of using Backup/Abstract for asynchronous fallback [11]. The comparison indicates that Transformer is much cheaper in contrast to the high cost of MVBA pace-sync. For example, Transformer always costs less than 1 second in BDT-sRBC, despite n and on/off of crashes, while MVBA pace-sync is about 10 times slower.

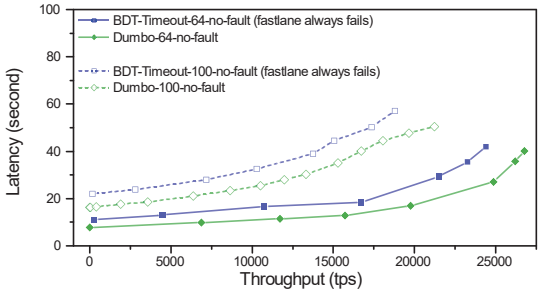

Figure 17: Latency v.s. throughput for experiments of BDT **with idling fastlane (i.e., fastlane just timeouts after 2.5 sec).**

As illustrated in Fig. 17, we measure the latency-throughput tradeoffs for BDT-Timeout, namely, to see how BDT worse than Dumbo when BDT's fastlane is under denial-of-service. This is arguably the worst-case test vector for BDT, since relative to Dumbo, it always costs extra 2.5 seconds to timeout and then executes the Transformer subprotocol. Nevertheless, the performance of BDT is still close to Dumbo. In particular, to realize the same throughput, BDT spends only a few additional seconds (which is mostly caused by our conservation 2.5-second timeout parameter).

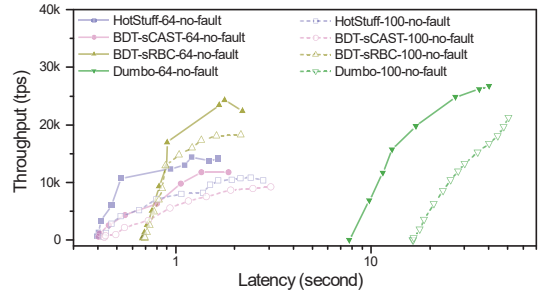

Figure 18: Throughput v.s. latency for experiments over WAN when n = 64 and 100, respectively (in case of periodically running pace-sync in BDT **per only 50 fastlane blocks).**

Latency-throughput trade-off. Figure 18 plots latency-throughput trade-offs of BDT-sCAST, BDT-sRBC, HotStuff and Dumbo in the WAN setting for n= 64 and 100 parties. This illustrates that BDT has low latency close to that of HotStuff under varying system load.

Either BDT-sCAST or BDT-sRBC is much faster than Dumbo by several orders of magnitude in all cases, while the two BDT instantiations have different favors towards distinct scenarios. BDT-sCAST has a latency-throughput trade-off similar to that of 2-chain HotStuff, and their small variance in latency is because we intentionally trigger timeouts in BDT-sCAST after each 50 fastlane blocks. BDT-sRBC has a latency-throughput trend quite different from HotStuff and BDT-sCAST. Namely, when fixing larger throughput, BDT-sRBC has a latency less than BDT-sCAST's; when fixing small throughput, BDT-sRBC could be slower. This separates them clearly in terms of application scenarios, since BDT-sRBC is a better choice for large throughput-favoring cases and BDT-sCAST is more suitable for latency-sensitive scenarios.[7]

[6] Following [11] that used full-fledged SMR to instantiate Backup for fallback, one can combine stable-leader 2-chain HotStuff (the fastlane of BDT-sCAST) and Dumbo by a single block of asynchronous SMR. This intuitive idea can be realized from MVBA [24, 69] as follows after the fastlane times out: each party signs and multicasts the highest quorum proof received from HotStuff, then waits for $n-f$ such signed proofs from distinct parties, and takes them as MVBA input; MVBA thus would output $n-f$ valid HotStuff quorum proofs (signed by $n-f$ parties), and the highest quorum proof in the MVBA output can represent the HotStuff block to continue Dumbo.

[7] Note that we can expect the throughput-latency tension in all protocols to be alleviated once adapting the recent mempool [35, 42] into their implementations.

Summary of evaluations in the WAN setting. The above results clearly demonstrate the efficiency of our pace-synchronization—Transformer. And thanks to that, BDT in the WAN setting is:

- As fast as 2-chain HotStuff in the best case (i.e., synchronous network without faulty parties); [8]
- As robust as the underlying asynchronous pessimistic path in the worst case (i.e., the fastlane always completely fails).

7.2 Evaluation in controlled dynamic network

Setup on the simulated fluctuating network. We also deploy our Python-written protocols for n=64 parties in a high-performance server having 4 28-core Xeon Platinum 8280 CPUs and 1TB RAM. The code is same to the earlier WAN experiments, except that we implement all TCP sockets with controllable bandwidth and delay. This allows us to simulate a dynamic communication network.

In particular, we interleave "good" network (i.e., 50ms delay and 200Mbps bitrate) and "bad" network (i.e., 300ms delay and 50Mbps bitrate) in the following experiments to reflect network fluctuation. Through the subsection, BDT refers to BDT-sCAST, the approach of using Abstract primitive [11] to combine stable-leader 2-chain HotStuff (BDT-sCAST's fastlane) and Dumbo is denoted by HS+Abstract+Dumbo (where Backup/Abstract is instantiated by MVBA as explained in Footnote 5). For experiment parameters, the fastlane's timeout is set as 1 second, the fastlane block and pessimistic block contain 10^4 and 10^6 transactions respectively, and we would report the number of confirmed transactions over time in random sample executions.

Good network with very short fluctuations. We first examine in a network that mostly stays at good condition except interleaving some short-term bad network condition that lasts only 2 seconds (which just triggers fastlane timeout). The sample executions in the setting are plotted in Fig. 19. The result indicates that the performance of BDT does not degrade due to the several short-term network fluctuation, and it remains as fast as 2-chain HotStuff. This feature is because BDT adopts a two-level fallback mechanism, such that it can just execute the light pace-sync and then immediately retry another fastlane. In contrast, using Backup/Abstract primitive (instantiated by MVBA) as pace-sync would encounter rather long latency (~25 sec) to run the heavy pace-sync and pessimistic path after the short-term network fluctuations.

Intermittent network with long bad time. We then evaluate the effect of long-lasting bad network condition. We visualize such sample executions in Fig. 20. Clearly, BDT can closely track the performance of its underlying pessimistic path during the long periods of bad network condition. Again, this feature is a result of efficient pace-sync, as it adds minimal overhead to the fallback. In contrast, using Backup/Abstract primitive (instantiated by MVBA) to compose stable-leader HotStuff and Dumbo would incur a latency ~10 seconds larger than BDT during the bad network due to its cumbersome pace-sync.

Summary of evaluations in fluctuating network. As expected by our efficient pace-sync subprotocol, BDT also performs well in the fluctuating network environment. Specifically,

[8] As discussed in Footnote 4, BDT-sCAST's fastlane has a 5-round latency, which is same to that of 2-chain HotStuff. The tiny difference between their evaluated latency is because we periodically trigger Transformer in the experiments of BDT-sCAST.

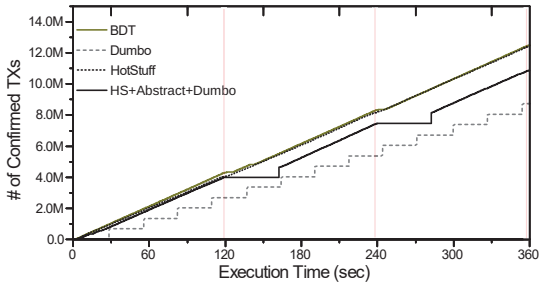

Figure 19: Sample executions of BDT, 2-chain HotStuff, Dumbo, and the composition of HotStuff+Abstract+Dumbo for n=64, when facing a few 2-second bad periods. The red region represents the 2-second period of bad network.

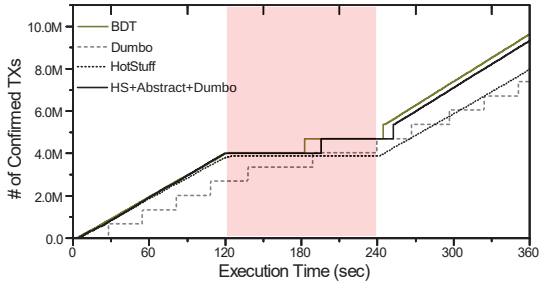

Figure 20: Sample executions of BDT, 2-chain HotStuff, Dumbo, and the composition of HotStuff+Abstract+Dumbo for n=64, when suffering from 120-second bad network. The red region represents the 120-second period of bad network.

- When encountering short-term network fluctuations, BDT can quickly finish pace-sync and restart a new fastlane, thus progressing at a speed same to 2-chain HotStuff.
- When the network becomes slow for longer periods (and even HotStuff grinds to a halt), BDT still is robust to progress nearly as fast as the underlying asynchronous protocol.

8 CONCLUSIONS

We propose the first *generic* and *practical* framework for optimistic asynchronous atomic broadcast BDT, in which we abstract a new and simple deterministic fastlane that enables us to reduce the asynchronous pace-synchronization to the conceptually minimum binary agreement. Several interesting questions remain: theoretically, it is interesting to formally demonstrate the efficiency gap between asynchronous and deterministic consensuses, e.g., better complexity lower bounds; practically, our current pessimistic path requires asynchronous common subset, and direct building from any asynchronous atomic broadcast may need further care; also, the paradigm of adding an optimistic path could be further generalized to provide not only efficiency but also better resilience or flexibility.

ACKNOWLEDGMENTS

We would like to thank Vincent Gramoli and the anonymous reviewers for their valuable comments. Yuan is supported in part by National Key R&D Project of China under Grant 2022YFB2701600, NSFC under Grant 62102404 and the Youth Innovation Promotion Association CAS. Qiang and Zhenliang are supported in part by research gifts from Ethereum Foundation, Stellar Foundation, Protocol Labs, Algorand Foundation and The University of Sydney.

REFERENCES

[1] Bug in ABA protocol's use of Common Coin. https://github.com/amiller/HoneyBadgerBFT/issues/59

[2] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. 2022. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement. In *Proc. PODC 2022*. 381–391.

[3] Ittai Abraham, Danny Dolev, and Joseph Y Halpern. An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In *Proc. PODC 2008*. 405–414.

[4] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. 2021. Reaching consensus for asynchronous distributed key generation. In *Proc. PODC 2021*. 363–373.

[5] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 106–118.

[6] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proc. PODC 2019*. 337–346.

[7] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *Proc. PODC 2021*.

[8] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2010. Prime: Byzantine replication under attack. *IEEE transactions on dependable and secure computing* 8, 4 (2010), 564–577.

[9] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of tendermint-core blockchains. In *Proc. OPODIS 2018*.

[10] Hagit Attiya and Jennifer Welch. 2004. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons.

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

[12] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. Rbft: Redundant byzantine fault tolerance. In *Proc. ICDCS 2013*. 297–306.

[13] Michael Ben-Or. Another advantage of free choice (Extended Abstract) Completely asynchronous agreement protocols. In *Proc. PODC 1983*. 27–30.

[14] Michael Ben-Or and Ran El-Yaniv. 2003. Resilient-optimal interactive consistency in constant time. *Distributed Computing* 16, 4 (2003), 249–262.

[15] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proc. PODC 1994*. 183–192.

[16] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. 2014. State machine replication for the masses with BFT-SMaRt. In *Proc. DSN 2014*. 355–362.

[17] Erica Blum, Jonathan Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Proc. TCC 2019*. 131–150.

[18] Erica Blum, Jonathan Katz, and Julian Loss. 2021. Tardigrade: An Atomic Broadcast Protocol for Arbitrary Network Conditions. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 547–572.

[19] Erica Blum, Chen-Da Liu-Zhang, and Julian Loss. 2020. Always have a backup plan: fully secure synchronous MPC with asynchronous fallback. In *Annual International Cryptology Conference*. Springer, 707–731.

[20] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *Proc. PKC 2003*. 31–46.

[21] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

[22] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).

[23] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proc. CCS 2002*. 88–97.

[24] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proc. CRYPTO 2001*. 524–541.

[25] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous Byzantine agreement using cryptography. In *Proc. PODC 2020*. 123–132.

[26] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Proc. SRDS 2005*. 191–201.

[27] Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *Proc. DISC 2017*.

[28] Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. STOC 1993*. 42–51.

[29] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

[30] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *Proc. OSDI 1999*. 173–186.

[31] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proc. AFT 2020*. 1–11.

[32] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults.. In *Proc. NSDI 2009*, Vol. 9. 153–168.

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

[34] Tyler Crain. 2020. Two More Algorithms for Randomized Signature-Free Asynchronous Binary Byzantine Consensus with t< n/3 and O(n^2) Messages and O(1) Round Expected Termination. *arXiv preprint arXiv:2002.08765* (2020).

[35] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proc. EuroSys 2022*. 34–50.

[36] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous data dissemination and its applications. In *Proc. CCS 2021*. 2705–2721.

[37] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. 2022. Practical Asynchronous Distributed Key Generation. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2518–2534.

[38] Sisi Duan, Michael K Reiter, and Haibin Zhang. BEAT: Asynchronous BFT made practical. In *Proc. CCS 2018*. 2028–2041.

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

[40] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Asscktion for Computing Machinery* 32, 2 (1985), 374–382.

[41] Matthias Fitzi and Juan A Garay. 2003. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. 211–220.

[42] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-NG: Fast Asynchronous BFT Consensus with Throughput-Oblivious Latency. In *Proc. CCS 2022*.

[43] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Efficient Asynchronous Byzantine Agreement without Private Setups. In *Proc. ICDCS 2022*.

[44] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2021. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. *arXiv preprint arXiv:2106.10362* (2021).

[45] Rati Gelashvili, Lefteris Kokoris-Kogias, Alexander Spiegelman, and Zhuolun Xiang. 2021. Be Prepared When Network Goes Bad: An Asynchronous View-Change Protocol. *arXiv preprint arXiv:2103.03181* (2021).

[46] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Proc. EUROCRYPT 1999*. 295–310.

[47] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proc. EuroSys 2010*. 363–376.

[48] 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 *Proc. DSN 2019*. 568–580.

[49] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Speeding Dumbo: Pushing Asynchronous BFT Closer to Practice. In *The 29th Network and Distributed System Security Symposium (NDSS)*.

[50] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proc. CCS 2020*. 803–818.

[51] Aniket Kate and Ian Goldberg. Distributed key generation for the internet. In *Proc. ICDCS 2009*. 119–128.

[52] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *Proc. PODC 2021*. 165–175.

[53] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures.. In *Proc. CCS 2020*. 1751–1767.

[54] Klaus Kursawe and Victor Shoup. first announced in 2002. Optimistic asynchronous atomic broadcast. In *Proc. ICALP 2005*. 204–215.

[55] Julian Loss and Tal Moran. 2018. Combining Asynchronous and Synchronous Byzantine Agreement: The Best of Both Worlds. *IACR Cryptol. ePrint Arch.* 2018 (2018), 235.

[56] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2021. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft. *arXiv preprint arXiv:2103.09425* (2021).

[57] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proc. PODC 2020*. 129–138.

[58] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proc. CCS 2016*. 31–42.

[59] Atsuki Momose, Jason Paul Cruz, and Yuichi Kaji. 2020. Hybrid-BFT: Optimistically Responsive Synchronous Consensus with Optimal Latency or Resilience. *IACR Cryptol. ePrint Arch.* 2020 (2020), 406.

[60] Atsuki Momose and Ling Ren. Multi-Threshold Byzantine Fault Tolerance. In *Proc. CCS 2021*.

[61] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proc. PODC 2014*. 2–9.
[62] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[63] Rafael Pass and Elaine Shi. 2017. The sleepy model of consensus. In *Advances in Cryptology – ASIACRYPT 2017*. 380–409.
[64] Arpita Patra. Error-free multi-valued broadcast and Byzantine agreement with optimal communication complexity. In *Proc. OPODIS 2011*. 34–49.
[65] Arpita Patra, Ashish Choudhary, and Chandrasekharan Pandu Rangan. Simple and efficient asynchronous byzantine agreement with optimal resilience. In *Proc. PODC 2009*. 92–101.
[66] Torben Pryds Pedersen. A Threshold Cryptosystem without a Trusted Party. In *Proc. EUROCRYPT 1991*. 522–526.
[67] R. Pass, and E. Shi. 2018. Thunderella: Blockchains with optimistic instant confirmation. In *Proc. EUROCRYPT 2018*. 3–33.
[68] Michael O Rabin. 1983. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*. IEEE, 403–409.
[69] HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Proc. OPODIS 2005*. 88–102.
[70] Muhammad Saad, Afsah Anwar, Srivatsan Ravi, and David Mohaisen. 2021. Revisiting Nakamoto Consensus in Asynchronous Networks: A Comprehensive Analysis of Bitcoin Safety and ChainQuality. In *Proc. CCS 2021*. 988–1005.
[71] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the Optimality of Optimistic Responsiveness. In *Proc. CCS 2020*. 839–857.
[72] Alexander Spiegelman. 2021. In Search for an Optimal Authenticated Byzantine Agreement. In *Proc. DISC 2021*.
[73] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. SRDS 2009*. 135–144.
[74] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks. In *Proc. NSDI 2022*.
[75] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proc. PODC 2019*. 347–356.
[76] Haibing Zhang and Sisi Duan. PACE: Fully Parallelizable BFT from Reproposable Byzantine Agreement. In *Proc. CCS 2022*.