Towards Optimal and Practical Asynchronous Byzantine Fault Tolerant Protocols

Zhenliang Lu

THE UNIVERSITY OF
SYDNEY

Supervisor: Qiang Tang

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

School of Computer Science
Faculty of Engineering
The University of Sydney
Australia

August 2023

ABSTRACT

With recent advancements in blockchain technology, people expect Byzantine fault tolerant (BFT) protocols to be deployed more frequently in wide-area networks (WAN) as opposed to conventional in-house settings. Asynchronous BFT protocols, which do not rely on any form of timing assumption, are arguably robust in such a setting. Asynchronous BFT protocols have been studied since the 1980s, but these asynchronous BFT works mainly focus on understanding the theoretical limits and possibilities. Until the recent asynchronous BFT protocol, HoneyBadgerBFT (HBBFT), was proposed, the field received renewed attention.

Dumbo family, a series of our works on the asynchronous BFT protocols, significantly pushed those protocols towards practice. First, all complexity metrics are pushed down to asymptotically optimal, simultaneously. Second, we identify the bottleneck in the state of the art and revisit the design methodology, identifying and utilizing the right components, and optimizing the protocol structure in various ways. Last but not least, we also open the box and optimize the critical components themselves. The resulting protocols are indeed significantly more performant, the latest protocol can have 100K tps and a few seconds of latency at a reasonable scale. This thesis focuses on the latest three members of the Dumbo family. To begin, we solved an open problem by proposing an optimal Multi-valued validated asynchronous Byzantine agreement protocol. Next, we present Dumbo-NG to address the challenge of latency-throughput tension by redesigning the methodology of asynchronous BFT protocols. Another benefit of the new methodology is that it can conquer the censorship threat without extra cost. Furthermore, we consider a realistic environment and present Bolt-Dumbo Transformer (BDT), a generic framework for practical optimistic asynchronous BFT to achieve the ”best of both worlds” in terms of the advantages of deterministic BFT and randomized (asynchronous) BFT.

List of Publications

Authorship is in alphabetical order

1. Yuan Lu, Zhenliang Lu, Qiang Tang. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT. *29th ACM Conference on Computer and Communication Security*, **CCS 2022**.

2. Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, Zhenfeng Zhang. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT. *29th ACM Conference on Computer and Communication Security*, **CCS 2022**.

3. Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, Zhenfeng Zhang. Speeding Dumbo: Pushing Asynchronous BFT Closer to Practice. *The Network and Distributed System Security Symposium 2022*, **NDSS 2022**.

4. Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, Zhenfeng Zhang. Efficient Asynchronous Byzantine Agreement without Private Setups. *42nd IEEE International Conference on Distributed Computing Systems*, **ICDCS 2022**.

5. Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, Zhenfeng Zhang. Dumbo: Faster Asynchronous BFT Protocols. *27th ACM Conference on Computer and Communication Security*, **CCS 2020**.

6. Yuan Lu, Zhenliang Lu, Qiang Tang, Guiling Wang. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. *39th ACM Symposium on Principles of Distributed Computing*, **PODC 2020**.

Patents:

1. Xinlei Zhai, Qiang Tang, Zhenliang Lu, Jing Xu, Zhenfeng Zhang, Bingyong Guo. Systems and methods for establishing consensus in distributed communications. *(International publication number: WO 2021/226846 A1)*. **PCT Patent**

2. Xinlei Zhai, Qiang Tang, Zhenliang Lu, Jing Xu, Zhenfeng Zhang, Bingyong Guo. Communication systems and methods for validation of a transaction via consensus in a distributed network environment. *(International publication number: WO 2021/226843 A1)*. **PCT Patent**

3. Hao Cheng, Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, Zhenfeng Zhang. Permissioned asynchronous blockchian consensus with broadcasts running off consensus. *PCT Patent under review*

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Name: Zhenliang Lu

Signature:

Date:

Authorship Attribution Statement

This thesis contains published material in the following publications, in which the authorship is in alphabetical order and I am the lead author.

- Yuan Lu, **Zhenliang Lu**, Qiang Tang. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT. *29th ACM Conference on Computer and Communication Security*, **CCS 2022**.
- Yingzi Gao, Yuan Lu, **Zhenliang Lu**, Qiang Tang, Jing Xu, Zhenfeng Zhang. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT. *29th ACM Conference on Computer and Communication Security*, **CCS 2022**.
- Yuan Lu, **Zhenliang Lu**, Qiang Tang, Guiling Wang. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. *39th ACM Symposium on Principles of Distributed Computing*, **PODC 2020**.

My particular contributions to this have been:

- Chapter 3 of this thesis is published as [88].

  I designed the study, analyzed the security, and wrote the draft.
- Chapter 4 of this thesis is published as [63].

  I designed the study, analyzed the security, and wrote the draft.
- Chapter 5 of this thesis is published as [87].

  I designed the study, analyzed the security, and wrote the draft.

Name: Zhenliang Lu

Signature:

Date:

Attesting Authorship Attribution Statement

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Supervisor Name: Prof. Qiang Tang

Signature:

Date:

To my beloved family.

ACKNOWLEDGMENT

The six years that I spent as a PhD student were an exciting journey for me. Specifically, this journey starts in Newark, New Jersey, USA, and ends in Sydney, Australia. This is also a real journey for me in the sense that it provides me with the opportunity to experience a completely different life in two different foreign countries. I would like to take this opportunity to express my sincere gratitude to everyone who accompanied and help me on my journey. First of all, I would like to express my deepest gratitude to my supervisor Prof. Qiang Tang, who offered me a great opportunity to pursue this path. The wisdom and enthusiasm that Prof. Qiang Tang demonstrates have had a significant impact on me. My interest in Blockchain research was sparked by him, and I am very grateful to him for both his guidance and his help. Without his guidance and help, I would never be as confident in my future as I am today.

I also would like to express my sincere thanks to the members of the committee, Prof. Andrew Miller and Prof. Fan Zhang, as well as to Prof. Vincent Gramoli, who served as the committee convenor, for their wonderful support and insightful comments.

I am especially grateful to New Jersey Institute of Technology and The University of Sydney, which provided the great opportunities, necessary infrastructure and resources for my research.

I would like to express my gratitude to the members of our group for the insightful discussions they contributed to our study and for all of the helpful assistance they provided to me throughout the course of my doctoral studies. I also would like to express my gratitude to all of my co-authors for the numerous successful collaborations we have had, as well as for sharing and discussing research ideas with me. Furthermore, I would like to thank everyone I have met who has helped me in creating wonderful memories outside

of work. An incomplete list includes Dr. Long Chen, Dr. Yuan Lu, Dr. Songlin He, Dr. Hanwen Feng, Dr. Bo Pang, Ms. Yanan Li, Mr. Tian Qiu, Ms. Xinrui Zhang, Dr. Bingyong Guo, Ms. Yingzi Gao, Mr. Pouriya Zarbafian, Dr. Ming Jin, and more. I had a great time with all of you on this journey, and the wonderful time we had together will stay with me forever.

Lastly, my deep gratitude goes to my family that encouraged me on this path. I would like to thank my lovely family for their unconditional love and support.

TABLE OF CONTENTS

TABLE OF CONTENTS

(Continued)

Chapter Page

TABLE OF CONTENTS

(Continued)

Chapter Page

TABLE OF CONTENTS

(Continued)

Chapter Page

TABLE OF CONTENTS

(Continued)

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES

(Continued)

Figure **Page**

LIST OF FIGURES

(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Asynchronous BFT protocol

This section will explain why we are researching asynchronous BFT protocols.

Byzantine fault-tolerant (BFT) protocols, as a fundamental research field in distributed computing, were first proposed by Lamport et al. [82]. BFT protocols are also known as Byzantine agreement or BFT consensus, informally. Such protocols enable a group of untrusted nodes to reach a consensus, making it possible for a distributed system to deliver the correct service in spite of network latency and the failure of nodes. They provide both strong safety and liveness, thus having the potential to realize highly available web services.

In particular, the recent success of blockchain promotes the development of the BFT protocols, in order to serve as the basic infrastructure of mission-critical applications, like transactional databases and other financial services. For realizing a critical global infrastructure with high-security assurance and well-distributed trust, one might expect that set of mutually distrusting and geologically distributed nodes to collectively maintain it. Clearly, such decentralized infrastructure has to be implemented atop consensus protocols that are both secure and efficient in a real-world Internet setting, thus resulting in an unprecedented demand for studying practical BFT consensus in fluctuating and varying network environments.

BFT protocols have been studied based on different timing (or network) assumptions. A synchronous BFT protocol assumes that all values sent by honest nodes will be delivered to the recipients within a predetermined time period. A partially synchronous BFT protocol, on the other hand, relaxes this network requirement by allowing the time bound to exist but be unknown. An asynchronous BFT protocol is the least reliant on network assumptions

because it does not require such time assumptions to exist, instead, it just needs to ensure that all values will be delivered eventually.

Many existing efforts [5, 8, 26, 45, 59, 90] are aimed at making BFT protocols work well in an in-house deployment environment. Such an environment usually has a relatively "private" network with well-connected nodes that can ensure message delivery within a certain period. Therefore, these BFT protocols are designed based on synchronous or partially synchronous assumptions. Typically, Google Chubby [31] and Apache Kafka [13] are deployed to enable a small number of nodes on a local-area network to tolerate crash failures.

However, in wide-area network (WAN) environment, these (partially) synchronous BFT protocols become unsuitable. They provide no guarantee when messages might be arbitrarily delayed. Moreover, it is challenging to decide the time parameter when deploying these BFT protocols in WAN, and the protocols could be of no progress and would become stuck when the actual network delay is larger than the selected time parameter. It was recently demonstrated [92] that classic partial synchronous Practical Byzantine Fault Tolerance (PBFT) [41] cannot advance in "intermittently synchronous network", when the adversary only chooses to delay messages occasionally. Similar results might be obtained by applying the "attack" on a group of synchronous and partially synchronous BFT protocols [11, 16, 24, 48, 52, 122].

Blockchain technology, in general, broadens the application scenarios for BFT protocols. Given that nodes in a Blockchain system are typically dispersed across the globe, and that the open Internet (i.e., WAN) environment presents an adversarial setting where network latency among nodes could vary over time, it is challenging to guarantee that the network conditions are stable between any two nodes in the WAN environment. As a result, an asynchronous blockchain system becomes a superior option for WAN since it more accurately captures the real-world network environment without assuming bounded network delay. Therefore, eliminating the synchronization assumption entirely becomes increasingly desirable for both robustness and efficiency.

Besides asynchronous protocols can better reflect real-world network environments, another important reason why asynchronous protocols are advantageous is their efficiency, particularly a property known as "responsiveness". When designing a synchronous BFT protocol, the assumed upper bound of network latency is parameterized, which is normally chosen to be large so that the actual network latency is always smaller to ensure the synchrony assumption is met. Due to this, the efficiency of most synchronous BFT protocols is related to the assumed network latency upper bound. While responsiveness requires that performance be only related to actual network latency, the protocol does not rely on any timing assumptions, and progress is made as soon as messages are delivered. Furthermore, because no time-out mechanism is required, asynchronous protocols significantly simplify the engineering efforts when actually building the distributed system. In contrast, when designing a system that implements a synchronous or partial synchronous protocol, all kinds of ad-hoc, error-prone time-out mechanisms should be considered, which might cause the whole system more complicated.

1.2 Prior to the Dumbo family

In this section, we will elaborate on the related work of asynchronous BFT protocols and discuss the practical obstacles of using them in the real world.

Since the 1980s when Lamport, Pease, and Shostak proposed the concept of Byzantine agreement (BA) in their seminal paper, a lot of research has been done on the problem, and many of them focused on the enticing asynchronous setting. One seminal work on this topic is an impossible result proven by Fischer, Lynch and Paterson (which is known as FLP "impossibility") [60]. The seminal FLP "impossibility" states that no deterministic consensus protocol can be possible in asynchronous settings if one node crashes. Many attempts [6, 9, 21–23, 34, 35, 37, 40, 80, 95–97, 107, 111, 113] have been made to design randomized asynchronous protocols to circumvent the "impossibility" for over four decades. The asynchronous atomic broadcast is a key component of fault-tolerant distributed systems. In fact, any protocol developed to solve the atomic broadcast can also be used to solve the Byzantine agreement, and vice versa [80, 113]. Atomic

broadcast allows a group of n nodes (in this thesis, the terms "node" and "party" are used interchangeably) to agree on a set of values to deliver and also on their delivery order, despite the failure of up to f ($< n/3$) nodes. In other words, the atomic broadcast can be viewed as the continuous process of BA. Even though, most of those studies on asynchronous atomic broadcast (ABC) protocols focused on theoretical feasibility, unsurprisingly, they have prohibitively high costs and very few implementations have inferior performance. Specifically, Ben-Or [21] presents an asynchronous protocol that requires exponential time complexity. Later, Canetti and Rabin [40] propose a polynomial-time asynchronous Byzantine agreement protocol, but the message complexity blows up and becomes the bottleneck in their work.

In the 2000s, the work of Cachin et al. [35] considers computationally bound adversaries and a trusted setup, which for the first time achieves a fairly practical polynomial-time protocol for asynchronous Binary Byzantine agreement via public-key cryptographic primitives. Based on [35], Cachin et al. [34] present a fairly practical protocol for asynchronous atomic broadcast, but the (per message) communication complexity is still up to $O(n^3)$ if there are n nodes. After that, two new asynchronous atomic broadcasts are proposed [80, 113]. They consider when the actual network delay could be reasonable, and at which point their protocol will work with the optimistic phase. Otherwise, it will automatically shift the protocol into the pessimistic phase via a pace-synchronization mechanism (analog to view-change with asynchronous securities). The optimistic cost in [80] is $O(n^2)$ communication complexity, and afterward, the work of [113] reduces the communication complexity from $O(n^2)$ to $O(n)$ in the optimistic phase. Even so, the pessimistic phase and pace-synchronization are still too heavy, which is due to the cost of communication complexity still being up to $O(n^3)$. As a result, it is fundamentally challenging to realize practical asynchronous BFT consensus, and none of such protocols has been widely adopted due to serious efficiency concerns, e.g., these BFTs [9, 23, 34, 37, 80, 113] have a high communication complexity, and the performance of these protocols will drop sharply when the system scales up.

Although the majority of earlier studies on asynchronous BFT were theoretical in nature, researchers continued to make great efforts and finally saw the light when the first practical asynchronous atomic broadcast protocol, HBBFT, was proposed in [92]. In order to push asynchronous BFT towards being practical, the elegant work of HBBFT observes that asynchronous atomic broadcast could be very lightly built from a weaker variant called asynchronous common subset (ACS) together with a threshold encryption scheme. The ACS protocol from [23, 92] was made up of two sub-protocols: reliable broadcast (RBC) and asynchronous binary agreement (ABA). An ACS protocol only requires nodes to agree on a subset of all their inputs. More importantly, it was observed in [92] that the classic ACS protocol from Ben-Or et al. [23] is much more likely to be efficient, both in terms of asymptotic communication complexity and practically, if its building blocks are carefully chosen. Even though there is a way to build ACS from MVBA [34], unfortunately, the only construction [34] brings huge overhead in communication complexity, and the MVBA primitive becomes a bottleneck in this way of construction.

Compared with the work of [34], although the communication complexity of HBBFT is optimal in theoretical, the round complexity blows up to $O(\log n)$ from constant in [34]. Hence, reducing the round complexity of $O(\log n)$ to the constant could increase practical performance.

1.3 Evolving of the Dumbo family

We will elaborate on the works of the Dumbo family in this section. These works also reflect the evolution of the Dumbo family.

In our series of works on the Dumbo family [63, 73, 74, 87, 88] , our first work, Dumbo BFT protocol [74], first identified the major bottleneck of HBBFT and reduced the number of ABA instances by re-introducing MVBA into ACS, finally achieving the constant running time that dramatically improved the performance.

Second, in Dumbo BFT, there are two reasons contributing to the inferior performance: (1) The RBC protocols still incur substantial costs; (2) the MVBA protocol itself is quite

complicated and heavy. So focusing on these pain points, we present Speeding Dumbo [73], a new member of the Dumbo family, that replaces the RBC instance with a cheaper broadcast component. Moreover, it also proposed a new MVBA protocol, Speeding MVBA (sMVBA), which is concretely more efficient than all existing MVBAs (at the time of proposing it).

Third, our other Dumbo family member, Dumbo-MVBA [88], answers a nearly 20-year open problem to asymptotically reduce the communication complexity from $O(\ell n^2 + \lambda n^2 + n^3)$ [34] to $O(\ell n + \lambda n^2)$ (where n is the number of parties, ℓ is the input length, and λ is the security parameter), which helps MVBA for the first time achieve optimal communication complexity when $\ell > \lambda n$.

Fourth, with careful analysis of these many existing asynchronous protocols (e.g. HBBFT and Dumbo [73, 74]), we can find that these protocols consist of two main phases—the bandwidth-intensive transaction dissemination phase and the bandwidth-oblivious agreement phase. The dissemination phase is bandwidth-intensive, however, the agreement phase is bandwidth-oblivious and latency-critical. Hence, it is unprecedentedly urgent to implement a robust BFT consensus that realizes high throughput while preserving low latency. Our Dumbo family member, Dumbo-NG [63], is a novel asynchronous BFT atomic broadcast designed to address these practical issues.

Fifth, the last member of the Dumbo family considered the practical issues when deploying these asynchronous atomic broadcast protocols in WAN. We all know that the actual WAN network is always changing. Because the Internet is so dynamic, it creates new fundamental challenges for making BFT consensus protocols that are both secure and very efficient. In fact, when network conditions are good, state-of-the-art randomized asynchronous consensus still performs much worse than deterministic (partial) synchronous consensus, especially when it comes to the critical latency metric. For this practical pain point, we present our last member, Bolt-Dumbo Transformer (BDT) [87], a generic framework for practical optimistic asynchronous atomic broadcast.

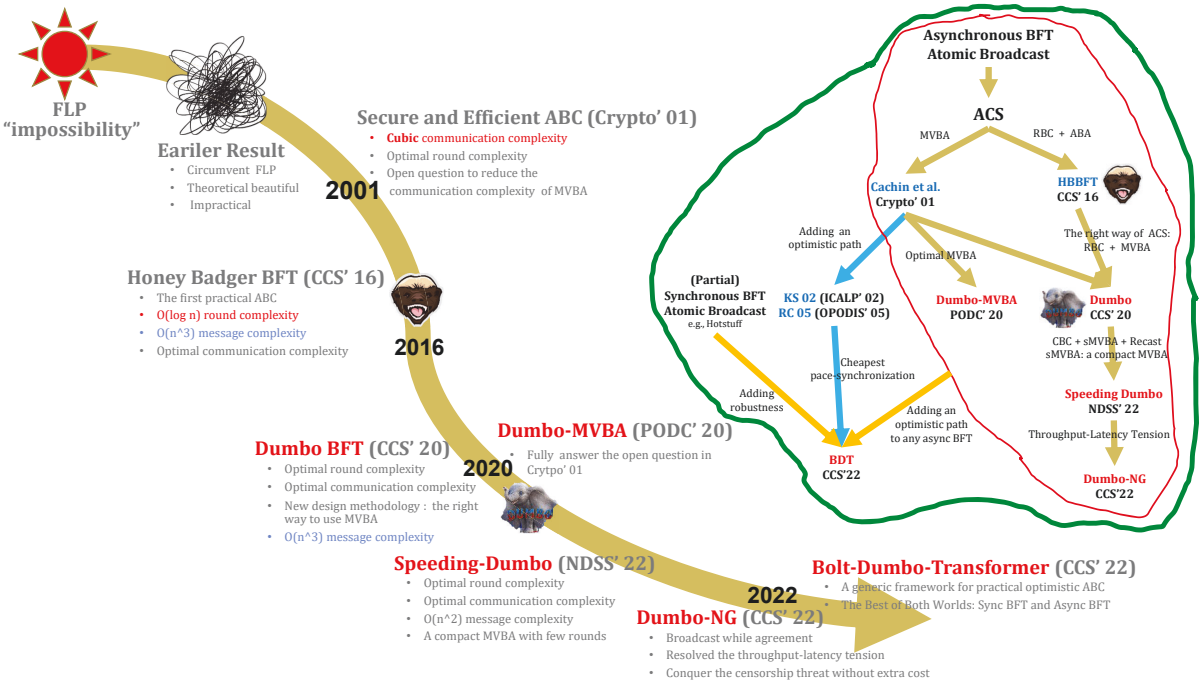


Figure 1.1: The evolution of the Dumbo family.

As shown in Figure 1.1, the Dumbo family evolves step by step with the clear aim of achieving a more practical asynchronous BFT consensus. We elaborate on the work of each member as follows:

1. **Dumbo BFT**. As the first practical asynchronous atomic broadcast protocol, HBBFT proposed by Miller et al. in [92], demonstrated impressive performance. The core of HBBFT is to achieve batching consensus using ACS of Ben-Or et al., constituted with n RBC to have each node propose its input, followed by n ABA to make a decision for each proposed value. Due to the famous FLP impossibility [60], an ABA must be a randomized protocol. This brings in the following drawback: though the expected number of "rounds" of each ABA protocol is constant, the expected number of rounds of running n concurrent ABA instances could be significant, i.e., at least $O(\log n)$ [22]. More seriously, those ABA instances do not really execute in a fully concurrent fashion. When n gets larger, and the network is unstable, there will likely be some ABA instances that terminate very slowly. The slowest ABA instance dominates the running time of the ACS of HBBFT. The practical impact of ABA

protocols on the performance of HBBFT was shown in Figure 2 of [74], it is clear that for HBBFT, the cost of ABA is dominating, and the pattern becomes even more significant as the scale of the system grows. This simple observation inspires us to reduce the number of ABA instances needed in the ACS protocol.

In Dumbo BFT, we propose two new atomic broadcast protocols (called Dumbo1, Dumbo2) both of which have asymptotically and practically better efficiency. In particular, the ACS of Dumbo1 only runs a small κ (independent of n) instances of ABA, while that of Dumbo2 further reduces it to constant! At the core of our techniques are two major observations: (1) reducing the number of ABA instances significantly improves efficiency; and (2) using MVBA which was considered sub-optimal for ACS in [92] in a more careful way could actually lead to a much more efficient ACS.

2. **Speeding Dumbo**. Despite the work of the Dumbo BFT protocol, which redesigned the HBBFT protocol backbone that used one MVBA to replace n concurrent ABA protocols and dramatically improved the performance, asynchronous BFT protocols remain slow, and in particular, the latency is still quite large. There are two reasons contributing to the inferior performance: (1) The RBC protocols still incur substantial costs; (2) the MVBA protocols are quite complicated and heavy, and all existing constructions need dozens of rounds and take the majority of the overall latency.

   We propose Speeding Dumbo (sDumbo) to continue pushing forward the performance of asynchronous BFT consensus protocols: we first reduce the message complexity to optimal via replacing the RBC instance with a cheaper broadcast component; together with the recent major progresses [74, 92], we have an asynchronous consensus protocol that is optimal for all major metrics, including round complexity, communication complexity (when the batch size is moderate), and message complexity. We then design a compact MVBA protocol (dubbed sMVBA, that can be of independent interest and use), so that its round about complexity is reduced from multiple dozens to a dozen or fewer. It requires only 6 rounds in

the best case and is expected to require 12 rounds in the worst case (by contrast, several dozens of rounds were required in the MVBA from Cachin et al. [34] and the Dumbo-MVBA [88], and around 20 rounds in the MVBA from Abraham et al. [9]).

3. **Dumbo-MVBA**. Typically, multi-valued asynchronous Byzantine agreement instances with external validity are executed sequentially to instantiate asynchronous atomic broadcast [34, 49]. MVBA, proposed in the elegant work of Cachin et al. [34], is fundamental for critical fault-tolerant services such as ABC in the asynchronous network. For example, MVBA was used as a core building block to implement ABC [33, 80, 113]. In MVBA, each party takes a value as input and decides one of the values as output, as long as the decided output satisfies the external validity condition. It was left as an open problem to asymptotically reduce the $O(\ell n^2 + \lambda n^2 + n^3)$ communication in [34] (where n is the number of parties, ℓ is the input length, and λ is the security parameter). This is also the reason why the HBBFT uses the ACS of Ben-Or et al. [23], which is composed of n RBC and n ABA, instead of using ACS of Cachin et al. [34], which reduces ACS to MVBA. Even if the former ACS has $O(logn)$ time complexity, while the latter ACS is constant. Recently, Abraham et al. [9] removed the n^3 term to partially answer the question when input is small. However, in other typical cases, e.g., building atomic broadcast through MVBA, the input length $\ell \geq \lambda n$, and thus the communication is dominated by the ℓn^2 term and the problem raised by Cachin et al. remains open.

   We present two MVBA protocols (Dumbo-MVBA and Dumbo-MVBA⋆) that reduce the communication cost of prior art [9, 34] by an $O(n)$ factor. At the core of our design, we propose asynchronous provable dispersal broadcast (APDB) in which each input can be split and dispersed to every party and later recovered in an efficient way. First, leveraging APDB and asynchronous binary agreement, we design an optimal MVBA protocol, Dumbo-MVBA. Second, we also present a general self-bootstrap framework Dumbo-MVBA⋆ that can reduce the communication of any existing MVBA protocols. The key idea is to invoke the underlying MVBA with taking as input the small-size proofs of APDB. Though Dumbo-MVBA⋆ is a "reduction" from

MVBA to MVBA itself, an advanced module instead of more basic building block such as binary agreement, this self-bootstrap technique can better utilize MVBA to achieve a simple modular design. These communication-efficient MVBA protocols also attain other optimal properties, asymptotically. Our results complement the recent breakthrough of Abraham et al. at PODC '19 [9] and solve the remaining part of the long-standing open problem from Cachin et al. at CRYPTO '01 [34].

Finally, we show that our MVBA protocols can immediately be applied to construct efficient asynchronous atomic broadcast with reduced communication blow-up as previously suggested in [34]. Moreover, they can provide better building blocks for the Dumbo BFT protocols [63, 73, 74], the recent constructions of practical asynchronous atomic broadcast that rely on MVBA at their heart for efficiency.

4. **Dumbo-NG**. It is unprecedentedly urgent to implement a robust BFT consensus that realizes high throughput while preserving low latency. In order to achieve their maximum throughput, many state-of-the-art asynchronous protocols typically sacrifice latency, which presents a serious practical challenge. To see the reason behind the throughput-latency tension in the existing performant asynchronous BFT protocols (with linear amortized communication complexity) such as HBBFT/Dumbo, recall that these protocols consist of two main phases—the bandwidth-intensive transaction dissemination phase and the bandwidth-oblivious agreement phase. The dissemination phase is bandwidth-intensive as it exchanges a large volume of transactions. However, the agreement phase is bandwidth-oblivious and latency-critical, as it need to exchange many rounds of short messages to decide which broadcasts shall be part of the output. Different from the dissemination phase that contributes into throughput, the agreement phase just hinders throughput, as it "wastes" available bandwidth in the sense of incurring large latency to block successive epochs' broadcasts. Thus, to sustain high throughput, each node has to broadcast a huge batch of transactions to "contend" with the agreement phase to seize most available bandwidth resources. However, larger batches unavoidably

cause inferior latency, although they can saturate the network capacity to obtain the maximum throughput.

Besides the unpleasant throughput-latency tension, the asynchronous protocols might also face serious censorship threat. This is because during the transaction dissemination phase, the adversarial network can delay the broadcasts containing its disliked transactions and prevent the certain transactions from being output. To mitigate the censorship threat and ensure liveness, existing designs rely on asymptotically larger communications, costly cryptographic operations, or probably unbounded memory. As a result, existing asynchronous BFT atomic broadcast protocols pose the next challenge, how to achieve the asynchronous BFT consensus further to realize minimum latency, maximum throughput, and guaranteed censorship-resilience, simultaneously?

We discuss the pain points of the current asynchronous BFT atomic broadcast, and then presents Dumbo-NG. The core of Dumbo-NG is to separate the message broadcast and consensus process, the consensus process works aside from the broadcast process. Dumbo-NG is a novel asynchronous BFT atomic broadcast protocol designed to address the remaining practical issues. Its technical core is a non-trivial direct reduction from asynchronous atomic broadcast to MVBA with quality property (which ensures the output of MVBA is from honest nodes with 1/2 probability). Dumbo-NG deconstructs the prior broadcast-then-consensus paradigm, and makes the transaction broadcasts completely running off consensus. Most interestingly, the new protocol structure empowers completely concurrent execution of transaction dissemination and asynchronous agreement. This brings about two benefits: (i) the throughput-latency tension is resolved to approach peak throughput with minimal increase in latency; (ii) the transactions broadcasted by any honest node can be agreed to output, thus conquering the censorship threat without extra cost.

5. **Bolt-Dumbo Transformer**. 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. Unfortunately, these synchrony assumptions may not always hold in the wide-area network, because of fluctuating bandwidth, unreliable links, substantial delays, and even network attacks. In contrast to the deterministic partially-synchronous or synchronous protocols, asynchronous BFT can ensure liveness and responsiveness without any form of timing assumptions. The above issues correspond to a fundamental "dilemma" lying in the design space of BFT consensus protocols suitable for the open Internet: 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. Unfortunately, existing works [80, 113] try to address this issue, but they directly use a heavy tool of 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. This design enables a highly efficient pace-synchronization to handle fallback. Specifically, BDT features low latency (same to the state-of-the-art pipelined consensus) when the network remains in good conditions for most of the time, and it can closely track the performance of the underlying fully asynchronous protocol when facing much worse network conditions. The resulting design reduces a cumbersome MVBA to a variant of the conceptually simplest binary agreement only. Besides, to our knowledge, this is the first optimistic asynchronous atomic broadcast realizing such a level of applicability, because of a

highly efficient pace-synchronization subprotocol that is reduced to the conceptually minimum binary agreement.

The goal of the Dumbo family is to design asymptotically optimal and practical asynchronous Byzantine fault tolerant protocols. We would strive to provide asymptotic improvements over existing asynchronous protocols, as well as practical optimizations such that the end protocols can indeed be deployed in a real-world environment. We believe that the Dumbo family could be a promising outcome toward this goal and might be a major step forward in asynchronous BFT protocols.

1.4 Recent Developments on DAG-based Asynchronous BFT

Apart from the previously mentioned asynchronous BFT approach, there's an alternative method to achieve asynchronous BFT by utilizing a directed acyclic graph (DAG). For ease of expression, we'll call this type of asynchronous BFT as DAG-based asynchronous BFT.

In DAG, each vertex represents a message sent by a node (sender), and each message carries transactions and references. References indicate that the sender received vertices in the last round. Those references form the edges of the DAG. The most significant benefit of the DAG is its elegant data structure, which contains information that enables nodes to totally order the DAG from their own perspective, all without needing to send additional messages. In other words, after constructing the DAG, achieving consensus on it doesn't involve any additional communication cost. This effective data structure has drawn the interest of numerous researchers who utilize DAG to design asynchronous protocols [19, 53, 54, 69, 70, 77, 98, 103, 109, 116]. The earliest instance of employing the DAG concept to devise asynchronous protocols can be traced back to 1999 [98]. Since then, very few papers recognized the potential of using DAGs to construct asynchronous BFT until the proposal of HashGraph in 2016 [19] and subsequently Aleph in 2019 [70]. However, their performance continues to be unsatisfactory. To be more specific, Hashgraph was constructed using an unstructured DAG and a local coin approach to circumvent the

FLP impossibility [60]. Unfortunately, this approach could result in high communication costs and exponential time complexity. Aleph achieves an expected constant latency. It does this by utilizing a structured, round-based DAG and incorporating shared randomness in each round. However, it's worth noting that Aleph needs to rely on unbounded memory, as explained in [54, 69].

Recent advancements in DAG-based asynchronous Byzantine atomic broadcast include efforts like DAG-Rider [77], Tusk [54] and Bullshark [69]. DAG-Rider achieves three significant objectives. First, it has optimal amortized communication complexity by combining batching techniques with an efficient asynchronous verifiable information dispersal protocol [38]. Second, it assures post-quantum safety by avoiding reliance on cryptographic primitives that could be compromised by quantum computers. Finally, DAG-Rider ensures eventual fairness, ensuring that all transactions proposed by honest nodes are eventually delivered. Tusk decouples transaction diffusion and agreement in an asynchronous setting, allowing consensus to concentrate on ordering small-size references, such that the approach yields unexpected system throughput. However, due to the fundamental technical aspect of Tusk involving a garbage collection mechanism, the strict guarantee of eventual fairness is somewhat relaxed.

Despite achieving optimal complexity, both DAG-Rider and Tusk's results still possess several gaps that need to be resolved before practical deployment. Primarily, these protocols perform notably worse than deterministic synchronous protocols like [5, 8, 45, 125] during favorable network conditions, particularly in terms of latency. Additionally, there is currently no existing DAG-based asynchronous BFT protocol that can ensure eventual fairness while not requiring unbounded memory. The state-of-the-art DAG-based asynchronous BFT protocol is Bullshark, and it effectively addresses all these challenges. To begin with, it's tailored for the usual synchronous scenarios, offering a low latency fast-path. Furthermore, Bullshark maintains the desired properties introduced in protocols like DAG-Rider while also demonstrating impressive practical performance.

In the previous section where we introduced asynchronous BFT, we referred to it as BA-based asynchronous BFT. In contrast to BA-based asynchronous BFT, DAG-based asynchronous BFT is a diverging paradigm, and all of them are pipelined BFT. DAG-based asynchronous BFT protocols are concurrently developed alongside BA-based asynchronous BFT, potentially enjoying better parallelism but also introducing other trade-offs. Thus, the practical superiority of either approach remains uncertain and requires further investigation. In this thesis, we only focus on BA-based asynchronous BFT.

1.5 Structure of this Thesis

The thesis gives a brief overview of the two early members, Dumbo BFT and Speeding Dumbo, but doesn't go into detail about them. The main body of this thesis is the three most recent asynchronous protocols. In particular, this thesis is put together in the following way:

Chapter 2 introduces all relevant notations that we utilized in the protocols. In addition, it also introduces some needed cryptographic primitives and underlying asynchronous consensus protocols that can be employed as building blocks.

Chapter 3 discusses Dumbo-MVBA and its application to optimal asynchronous atomic broadcast. First, we propose asynchronous provable dispersal broadcast (APDB), and then use it to design an optimal MVBA protocol. After that, we present a general framework Dumbo-MVBA$\star$. In the end, we elaborate how to obtain efficient asynchronous atomic broadcast through improving Cachin et al.'s results in [34].

Chapter 4 presents Dumbo-NG, a fast asynchronous BFT consensus with throughput-oblivious latency. In this chapter, we point out that many recent efforts [73, 74, 92] insist on the broadcast-then-consensus paradigm, which is why these asynchronous BFT atomic broadcasts have suboptimal performance. Then, we break down the old broadcast-then-consensus model and make the transaction broadcasts run completely concurrent with the consensus process in a concise and efficient way. Finally, we evaluate our Dumbo-NG

and compare it to the state-of-the-art asynchronous BFT protocols Dumbo BFT and Speeding-Dumbo.

Chapter 5 discusses how to make a BFT consensus achieving the best of both synchronous and asynchronous paradigms, and proposes a generic framework Bolt-Dumbo Transformer (BDT). The outcome of chapter 5 is also can be directly applied to the consensus protocol of chapter 4. At the end of the chapter, we implement the related experiment to show the superiority of our proposed approach.

In the final Chapter 6 of this thesis, we provide a brief overview of this thesis, summarizing its main results, as well as discussing some potential studies that are promising and meaningful for future work.

CHAPTER 2

PRELIMINARIES

In this chapter, we introduce relevant notations, formal cryptographic primitives, and asynchronous consensus protocols that will be employed throughout this thesis.

2.1 Notations

We let $[n]$ be short for $\{1, 2, \ldots, n\}$. The cryptographic security parameter is denoted by λ, capturing the bit-length of signatures and hashes. We let $|B|$ denote the batch size parameter, i.e., each node always chooses $|B|$ transactions from its buffer to disseminate. $\langle x, y \rangle$ denotes a string concatenating two strings x and y. Throughout the paper, P.P.T. is short for probabilistic polynomialtime.

Any message between two parties is in the form of $\textsc{MsgType}(\mathsf{id}, \cdot, \cdot, \cdots)$, where $\textsc{MsgType}$ specifies the message type and id is the identifier tagging the protocol instance. We might omit the id field in messages for brevity if there is a clear context, and also let "A multicasts $\textsc{Msg}$" denote that a party A sends a message $\textsc{Msg}$ to all parties, and let "B aborts" denote that a party B quits the protocol execution. The pseudocode of our protocol follows the conventional way to describe asynchronous protocols [34, 88]: we let "*wait for*" to represent that the protocol is blocking until a certain event; sometimes, we omit an implicit "*wait for*" handler for the incoming messages and use "*upon receiving*" to describe how to process an applicable message; "*multicast*" means to send a message to all parties; "*output*" means to return some value without halt; "*return W*" means to output W and then halt; Moreover, $\Pi[\mathsf{id}]$ refers to an instance of some protocol Π with an identifier id, and $y \leftarrow \Pi[\mathsf{id}](x)$ means to invoke $\Pi[\mathsf{id}]$ on input x and wait for its output y.

Performance metrics. We are particularly interested in constructing practical asynchronous BFT protocols. So it becomes meaningful to consider the following key efficiency metrics:

- *(Amortized) communication complexity*. the expected number of bits exchanged among honest nodes for each output transaction;
- *Message complexity*. This characterizes the number of messages exchanged among honest parties to produce a block;
- *Asynchronous round complexity*. the expected asynchronous rounds needed to output a transaction tx (after an honest node invokes the protocol to totally order tx). Here asynchronous round is the "time" measurement in an asynchronous network, and can be viewed as a "time" unit defined by the longest delay of messages sent among honest nodes [40, 77].

2.2 Cryptographic Primitives and Protocols

Negligible function. We say that a function $negl : \mathbb{N} \to \mathbb{R}$ is negligible if for every positive integer c there exists an integer N_c such that for all $x > N_c$,

$$|negl(x)| < \frac{1}{x^c}.$$

Negligible probability and Overwhelming probability. If the probability of an event is a negligible function (in λ), it is said that the event happens with negligible probability; If the probability of an event happens except with negligible probability, the event is said with overwhelming probability.

Hash function. Hash functions $\mathcal{H}$ are functions that convert an input string of arbitrary length to a string of fixed length. Formally, a collision resistant hash function $\mathcal{H}$ [110] satisfying the following conditions:

1. The input string X can be of arbitrary length and the result $\mathcal{H}(X)$ is a fixed length string;
2. The hash function must be one-way in the sense that given a Y in the image of $\mathcal{H}$, it is "hard" to find a string X such that $\mathcal{H}(X) = Y$, and given X and $\mathcal{H}(X)$, it is "hard" to find a string $X' \neq X$ such that $\mathcal{H}(X) = \mathcal{H}(X')$;

3. The hash function must be collision resistant: this means that it is "hard" to find two distinct strings that hash to the same result.

Erasure code scheme. A (k, n)-erasure code scheme [25] consists of a tuple of two deterministic algorithms Enc and Dec. The Enc algorithm maps any vector $\mathbf{v} = (v_1, \cdots, v_k)$ of k data fragments into an vector $\mathbf{m} = (m_1, \cdots, m_n)$ of n coded fragments, such that any k elements in the code vector $\mathbf{m}$ is enough to reconstruct $\mathbf{v}$ due to the Dec algorithm. More formally, a (k, n)-erasure code scheme has a tuple of two deterministic algorithms:

1. $\mathsf{Enc}(\mathbf{v}) \to \mathbf{m}$. On input a vector $\mathbf{v} \in \mathcal{B}^k$, this *deterministic* encode algorithm outputs a vector $\mathbf{m} \in \mathcal{B}^n$. Note that $\mathbf{v}$ contains k data fragments and $\mathbf{m}$ contains n coded fragments, and $\mathcal{B}$ denotes the field of each fragment.

2. $\mathsf{Dec}(\{(i, m_i)\}_{i \in S}) \to \mathbf{v}$. On input a set $\{(i, m_i)\}_{i \in S}$ where $m_i \in \mathcal{B}$, and $S \subset [n]$ and $|S| = k$, this *deterministic* decode algorithm outputs a vector of data fragments $\mathbf{v} \in \mathcal{B}^k$.

We require (k, n)-erasure code scheme is *maximum distance separable*, namely, the original data fragments $\mathbf{v}$ can be recovered from any k-size subset of the coded fragments $\mathbf{m}$, which can be formally defined as:

- **Correctness of erasure code**. For any $\mathbf{v} \in \mathcal{B}^k$ and any $S \subset [n]$ that $|S| = k$, $\Pr[\mathsf{Dec}(\{(i, m_i)\}_{i \in S}) = \mathbf{v} \mid \mathbf{m} := (m_1, \cdots, m_n) \leftarrow \mathsf{Enc}(\mathbf{v})] = 1$. If a vector $\mathbf{m} \in \mathcal{B}^n$ is indeed the coded fragments of some $\mathbf{v} \in \mathcal{B}^k$, we say the $\mathbf{m}$ is well-formed; otherwise, we say the $\mathbf{m}$ is ill-formed.

Instantiation. Through the paper, we consider a $(f + 1, n)$-erasure code scheme where $3f + 1 = n$. Besides, we emphasize the erasure code scheme would implicitly choose a proper $\mathcal{B}$ according to the actual length of each element in $\mathbf{v}$, such that the encoding causes only constant blow-up in size, namely, the bits of $\mathbf{m}$ are different from the bits of $\mathbf{v}$ by at most a constant factor. One well-known instantiation of such the primitive is due to Rabin [112].

Position-binding vector commitment (VC). For an established position-binding n-vector commitment (VC), there is a tuple of algorithms (VCom, Open, VerifyOpen). On input a vector $\mathbf{m}$ of any n elements, the algorithm VCom produces a commitment vc for the vector $\mathbf{m}$. On input $\mathbf{m}$ and vc, the Open algorithm can reveal the element m_i committed in vc at the i-th position while producing a short proof π_i, which later can be verified by VerifyOpen.

Formally, a position-binding VC scheme (without hiding) is abstracted as:

1. $\mathsf{VC.Setup}(\lambda, n, \mathcal{M}) \rightarrow pp$. Given security parameter λ, the size n of the input vector, and the message space $\mathcal{M}$ of each vector element, it outputs public parameters pp, which are implicit inputs to all the following algorithms. We explicitly require $\mathcal{M} = \{0,1\}^*$, such that one VC scheme can commit any n-sized vectors.

2. $\mathsf{VCom}(\mathbf{m}) \rightarrow (\mathsf{vc}; \textsc{Aux})$. On input a vector $\mathbf{m} = (m_1, ..., m_n)$, it outputs a commitment string vc and an auxiliary advice string Aux. We might omit Aux for presentation simplicity. Note we do not require the hiding property, and then let VCom to be a deterministic algorithm.

3. $\mathsf{Open}(\mathsf{vc}, m_i, i; \textsc{Aux}) \rightarrow \pi_i$. On input $m_i \in \mathcal{M}$, $i \in [n]$, the commitment vc and advice Aux, it produces an opening string π to prove that m_i is the i-th committed element. We might omit Aux for presentation simplicity.

4. $\mathsf{VerifyOpen}(\mathsf{vc}, m_i, i, \pi_i) \rightarrow 0/1$. On input $m_i \in \mathcal{M}$ and $i \in [n]$, the commitment vc, and an opening proof π, the algorithm outputs 0 (accept) or 1 (reject).

An already established VC scheme shall satisfy **correctness** and **position binding**:

- **Correctness**. An established VC scheme with public parameter pp is correct, if for all $\mathbf{m} \in \mathcal{M}^n$ and $i \in [n]$, $\Pr[\mathsf{VC.VerifyOpen}(\mathsf{vc}, m_i, i, \mathsf{VC.Open}(\mathsf{vc}, m_i, i, \textsc{Aux})) = 1 \mid (\mathsf{vc}, \textsc{Aux}) \leftarrow \mathsf{VC.VCom}(\mathbf{m})] = 1$.

- **Position binding**. An established VC scheme with public parameter pp is said position binding, if for any P.P.T. adversary $\mathcal{A}$, $\Pr[\mathsf{VC.VerifyOpen}(\mathsf{vc}, m, i, \pi) =$

$\mathsf{VC.VerifyOpen}(\mathsf{vc}, m', i, \pi') = 1 \wedge m \neq m' \mid (\mathsf{vc}, i, m, m', \pi, \pi') \leftarrow \mathcal{A}(pp)] < negl(\lambda)$, where $negl(\lambda)$ is a negligible function in λ.

Instantiation. There are a few simple solutions to achieve the above VC notion. An example is hash Merkle tree [91] where the commitment vc is $O(\lambda)$-bit and the openness π is $O(\lambda \log n)$-bit. Moreover, when given computational Diffie-Hellman assumption and collision-resistant hash function, there is a position-binding vector commitment scheme [43], s.t., all algorithms (except setup) are deterministic, and both commitment and openness are $O(\lambda)$ bits.

Relying on computational Diffie-Hellman (CDH) assumption and collision-resistant hash function, there is a construction due to Catalano et al. [43] that realizes the position-binding vector commitment as follows:

- $\mathsf{VC.Setup}(\lambda, n) \rightarrow pp$. Let $\mathcal{H}$ be a collision-resistant hash function $\mathcal{H} : \{0,1\}^* \rightarrow \mathbb{Z}_p$. Given λ, generate $\mathbb{G}$ and $\mathbb{G}_T$ that are two bilinear groups of prime order p with a bilinear map $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$; also choose g that is a random generator of $\mathbb{G}$. Randomly choose $(z_1, \cdots, z_n)$ from $\mathbb{Z}_p$, and for each $i \in [n]$, compute $h_i \leftarrow g^{z_i}$. For each $i, j \in [n]$ and $i \neq j$, compute $h_{i,j} = g^{z_i z_j}$. Set $pp = (g, \{h_i\}_{i\in[n]}, \{h_{i,j}\}_{i,j\in[n],i\neq j})$.
- $\mathsf{VCom}(\mathbf{m}) \rightarrow (\mathsf{vc}; \textsc{Aux})$. Compute $\mathsf{vc} = \prod_{i=1}^{n} h_i^{\mathcal{H}(m_i)}$ and $\textsc{Aux} = (\mathcal{H}(m_1), \cdots, \mathcal{H}(m_n))$. The output $\textsc{Aux}$ might be omitted in the paper for simplicity.
- $\mathsf{Open}(\mathsf{vc}, m_i, i; \textsc{Aux}) \rightarrow \pi_i$. Compute $\pi_i = \prod_{j=1,j\neq i}^{n} h_{i,j}^{\mathcal{H}(m_i)} = (\prod_{j=1,j\neq i}^{n} h_j^{\mathcal{H}(m_i)})^{z_i}$. We might omit the input $\textsc{Aux}$ for presentation simplicity.
- $\mathsf{VerifyOpen}(\mathsf{vc}, m_i, i, \pi_i) \rightarrow 0/1$. It checks whether $e(\mathsf{vc}/h_i^{\mathcal{H}(m_i)}, h_i) = e(\pi_i, g)$ or not.

Regarding the *size* of the commitment and the openness proof, it is clear that they contain only a single group element in $\mathbb{G}$, which corresponds to only $O(\lambda)$ bits and is independent to the length of each committed element. In addition, all algorithms run in polynomial time, and they (except the setup) are *deterministic*.

Digtal signature is an authentication mechanism that enables the creator of a message to attach a code that acts as a signature. Formally, a digital signature scheme is a triple of probabilistic polynomial time algorithms ($\mathsf{Ds.gen}, \mathsf{DS.sign}, \mathsf{DS.verify}$), satisfying:

1. $\mathsf{Ds.gen}(1^\lambda) \rightarrow (pk, sk)$. On input λ, it randomized generates a public key (pk), and a corresponding private key (sk), where λ is the security parameter;
2. $\mathsf{DS.sign}(sk, m) \rightarrow \sigma_i$. On input a message m and a secret key share sk_i, this deterministic algorithm outputs a signature σ;
3. $\mathsf{DS.verify}(m, (pk, \sigma)) \rightarrow 0/1$. Given a message m, signature σ and pk, this algorithms outputs 1 (accept) or 0 (reject).

We require an established digtal signature scheme to satisfy **correctness** and **secure**:

- **Correctness**. The correctness property requires that: for $\forall\ m$,

$$\Pr[\mathsf{DS.verify}(m, pk, \mathsf{DS.sign}(m, sk)|(pk, sk) \rightarrow \mathsf{Ds.gen}(1^\lambda))] = 1.$$

- **Secure**. Unless sk leaks, no P.P.T. adversary can produce valid signature except with negligible probability in λ.

Threshold encryption (TPKE**))** is a cryptographic primitive that allows any node to encrypt a value, such that the network nodes must work together to decrypt it. We consider only t correct decryption shares for a ciphertext, the plaintext can be recovered. It guarantees that the adversary learns nothing about the plaintext unless at least $t - f$ correct node reveals its decryption share. More formally, an TPKE scheme consists of five algorithms:

- $\mathsf{TPKE.Setup}(n, t, 1^\lambda) \rightarrow (\mathsf{PK}, \mathsf{VK}, \mathbf{SK})$: Takes as input the number of decryption servers n, a threshold t, where $1 \leq t \leq n$, and a security parameter λ. It generates a public key PK, a verification key VK, and $\mathbf{SK} = (sk_1, \cdots, sk_n)$ is a vector of n private key shares. Node $\mathcal{P}_i$ gets $\{\mathsf{PK}, \mathsf{VK}, sk_i\}$;
- $\mathsf{TPKE.Enc}(\mathsf{PK}, m) \rightarrow C$: Takes as input a public key PK and a message m. It outputs a ciphertext C;

- $\mathsf{TPKE.DecShare}(\mathsf{PK}, i, sk_i, C) \rightarrow \sigma_i$: Takes as input the public key PK, a ciphertext C, and private key sk_i. It produces the $i-$th share of the decryption;
- $\mathsf{TPKE.DecVerify}(\mathsf{PK}, \mathsf{VK}, C, \{i, \sigma_i\}) \rightarrow 0/1$: Takes as input the public key PK and verification key VK, a ciphertext C, and a decryption share σ_i. It outputs 1 (valid) or 0 (invalid);
- $\mathsf{TPKE.Dec}(\mathsf{PK}, \mathsf{VK}, C, \{i, \sigma_i\}_{i\in[t]}) \rightarrow m$: Takes as input the public key PK and verification key VK, a ciphertext C, and k decryption shares $\{i, \sigma_i\}_{i\in[t]}$. It outputs plaintext m (or $\perp$).

We require an established TPKE scheme to satisfy two **consistency** properties:

- For any ciphertext C, if $\sigma_i = \mathsf{TPKE.DecShare}(\mathsf{PK}, i, sk_i, C)$ where sk_i is the $i-th$ private key share in $\mathbf{SK}$, then $\mathsf{TPKE.DecVerify}(\mathsf{PK}, \mathsf{VK}, C, \{i, \sigma_i\}) = 1$.
- If C is the output of $\mathsf{TPKE.Enc}(\mathsf{PK}, m)$ and $S = \{i, \sigma_i\}_{i\in[t]}$ is a set of decryption shares $\sigma_i = \mathsf{TPKE.DecShare}(\mathsf{PK}, i, sk_i, C)$ for t distinct private keys in $\mathbf{SK}$, then we require that $\mathsf{TPKE.Dec}(\mathsf{PK}, \mathsf{VK}, C, \{i, \sigma_i\}_{i\in[t]}) = m$.

Non-interactive threshold signature (TSIG). Given an established (t, n)-threshold signature, each party $\mathcal{P}_i$ has a private function denoted by $\mathsf{SignShare}_{(t)}(sk_i, \cdot)$ to produce its "partial" signature, and there are also three public functions $\mathsf{VerifyShare}_{(t)}$, $\mathsf{Combine}_{(t)}$ and $\mathsf{VerifyThld}_{(t)}$, which can respectively validate the "partial" signature, combine "partial" signatures into a "full" signature, and validate the "full" signature. Note the subscript (t) denotes the threshold t through the paper. Formally, a non-interactive (t, n)-threshold signature scheme TSIG is a tuple of hereunder algorithms/protocols among n parties $\{\mathcal{P}_i\}_{i\in[n]}$:

1. $\mathsf{TSIG.Setup}(\lambda, t, n) \rightarrow (mpk, \mathbf{pk}, \mathbf{sk})$. Given t, n and security parameter λ, generate a special public key mpk, a vector of public keys $\mathbf{pk} = (pk_1, \cdots, pk_n)$, and a vector of secret keys $\mathbf{sk} = (sk_1, \cdots, sk_n)$ where $\mathcal{P}_i$ gets sk_i only.

2. $\mathsf{SignShare}_{(t)}(sk_i, m) \rightarrow \sigma_i$. On input a message m and a secret key share sk_i, this deterministic algorithm outputs a "partial" signature share σ_i. Note that the subscript (t) denotes the threshold t.

3. $\mathsf{VerifyShare}_{(t)}(m, (i, \rho_i)) \rightarrow 0/1$. Given a message m, a "partial" signature ρ_i and an index i (along with implicit input mpk and **pk**), this deterministic algorithm outputs 1 (accept) or 0 (reject).

4. $\mathsf{Combine}_{(t)}(m, \{(i, \rho_i)\}_{i \in S}) \rightarrow \sigma/\perp$. Given a message m and t indexed partial-signatures $\{(i, \rho_i)\}_{i \in S}$ (along with implicit input mpk and **pk**), this algorithm outputs a "full" signature σ for message m (or $\perp$).

5. $\mathsf{VerifyThld}_{(t)}(m, \sigma) \rightarrow 0/1$. Given a message m and an "aggregated" full signature σ (along with the implicit input mpk), this algorithms outputs 1 (accept) or 0 (reject).

We require an established TSIG scheme to satisfy **correctness**, **robustness** and **unforgeability**:

- **Correctness**. The correctness property requires that: (i) for $\forall\ m$ and $i \in [n]$, $\Pr[\mathsf{VerifyShare}_{(t)}(m, (i, \rho_i)) = 1 \mid \rho_i \leftarrow \mathsf{SignShare}_{(t)}(sk_i, m)] = 1$; (ii) for $\forall\ m$ and $S \subset [n]$ that $|S| = t$, $\Pr[\mathsf{VerifyThld}_{(t)}(m, \sigma) = 1 \mid \forall i \in S, \rho_i \leftarrow \mathsf{SignShare}_{(t)}(sk_i, m) \wedge \sigma \leftarrow \mathsf{Combine}_{(t)}(m, \{(i, \rho_i)\}_{i \in S})] = 1$.

- **Robustness**. No P.P.T. adversary can produce t valid "partial" signature shares, s.t. running Combine over these "partial" signatures does *not* produce a valid "full" signature, except with negligible probability in λ. Intuitively, robustness ensures any t valid "partial" signatures for a message must induce a valid "full" signature [118].

- **Unforgeability** (against adaptive adversary). The unforgeability can be defined by a threshold and adaptive version Existential UnForgeability under Chosen Message Attack game [83]. Intuitively, the unforgeability ensures that no P.P.T. adversary $\mathcal{A}$ that adaptively corrupts f parties ($f < t$) can produce a valid "full" signature except with negligible probability in λ, unless $\mathcal{A}$ receives some "partial" signatures produced by $t - f$ parties that are honest.

Instantiation. The above adaptively secure non-interactive threshold signature can be realize due to the construction in [83], in which each "partial" signatures ρ and every "full" signature σ are $O(\lambda)$ bits in length.

Probabilistically uniformly bounded statistic. Here we showcase the definition of uniformly-bounded statistic, which is a conventional notion [9, 34] to rigorously describe the performance metrics of fault-tolerant protocols under the influence of arbitrary adversary, such as the messages and communicated bits among honest parties.

Definition 1. ***Uniformly Bounded Statistic***. *Let X be a random variable representing a protocol statistic (for example, the number of messages generated by the honest parties during the protocol execution). We say that X is probabilistically uniformly bounded, if there exists a fixed polynomial $T(\lambda)$ and a fixed negligible function $\delta(k)$, such that for any adversary $\mathcal{A}$ (of the protocol), there exists a negligible function $\epsilon(\lambda)$ for all $\lambda \geq 0$ and $k \geq 0$,*

$$\Pr[X \geq kT(\lambda)] \leq \delta(k) + \epsilon(\lambda).$$

A probabilistically uniformly bounded statistic of a protocol performance metric X cannot exceed the uniform bound except with negligible probability, independent of the adversary. More precisely, this means, there exists a constant c, s.t. for any adversary $\mathcal{A}$, the expected value of X must be bounded by $cT(k) + \epsilon'(\lambda)$, where $\epsilon'(\lambda)$ is a negligible function.

Threshold common coin (Coin). A (t, n)-Coin is a protocol among n parties, through which any t honest parties can mint a common coin r uniformly sampled over $\{0, 1\}^\kappa$. The adversary corrupting up to f parties (where $f < t$) cannot predicate coin r, unless $t - f$ honest parties invoke the protocol. Formally, an established (t, n)-Coin scheme satisfies the following properties except with negligible probability in λ:

- **Termination.** Once t honest parties activate Coin, each honest party that activates Coin will output a common value r.
- **Agreement.** If two honest parties output r and r' respectively, then $r = r'$.

- **Unbiasedness.** Under the influence of any P.P.T. adversary, the distribution of the outputs of Coin is computationally indistinguishable from the uniform distribution over $\{0,1\}^\kappa$.

- **Unpredictability.** A P.P.T. adversary $\mathcal{A}$ who can adaptively corrupt up to f parties (e.g. $f < t$) cannot predicate the output of Coin better than guessing, unless $t - f$ honest parties activate Coin.

Instantiation. (t, n)-Coin can be realized from non-interactive (t, n)-threshold signature due to Cachin, Kursawe and Shoup [36] in the random oracle model. Moreover, it is immediately to generalize the Coin protocol in [36] against adaptive adversary [9, 86], if being given non-interactive threshold signature with adaptive security [83]. Throughout this thesis, when we claim to defend against an adaptive adversary, then we assume all relevant common coins are adaptively secure, and can be instantiated through the construction in [86], which incur $O(\lambda n^2)$ bits, $O(n^2)$ messages and constant $O(1)$ running time, where λ is the cryptographic security parameter.

Identity election. In our context, an identity Election protocol is a $(2f + 1, n)$-Coin protocol that returns a common value over $\{1, \cdots, n\}$. Throughout the paper, this particular Coin is under the descriptive alias Election, which is also a standard term due to Ben-Or and El-Yaniv [22].

Reliable broadcast (RBC). In RBC, there has a designated sender who aims to send its input to all parties, Formally, an RBC satisfies the next properties except with negligible probability:

- *Validity*. If the sender is honest and inputs v, then all honest parties output v;

- *Agreement*. The outputs of any two honest parties are same;

- *Totality*. If an honest party outputs v, then all honest parties output v.

Asynchronous binary agreement (ABA)**.** In an asynchronous binary agreement (ABA) protocol among n parties, the honest parties input a single bit, and aim to output a common

bit $b \in \{0, 1\}$ which shall be input of at least one honest party. Formally, an ABA protocol satisfies the properties except with negligible probability:

- **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.
- **Agreement**. If any two honest parties output b and b' receptively, then $b = b'$.
- **Validity**. If any honest party outputs a bit $b \in \{0, 1\}$, then at least one honest party takes b as input.

Instantiation. Many optimally-resilient ABA protocols [34, 36, 99] can be tuned against adaptive adversary, if the underlying common coin implementation is adaptively secure [9, 86]. In particular, the ABA secure against adaptive adversary controlling up to $\lfloor \frac{n-1}{3} \rfloor$ parties in [86], which attains expected $O(1)$ running time, asymptomatic $O(n^2)$ messages and $O(\lambda n^2)$ bits, where λ is the cryptographic security parameter.

Multi-valued validated Byzantine agreement (MVBA). MVBA [9, 34, 88] is a variant of Byzantine agreement with external validity, such that the participating nodes can agree on a value satisfying a publicly known predicate Q. Each node in the MVBA protocol takes a (probably different) value validated by a global predicate Q (whose description is known by the public) as input, and decides a value satisfying Q as the output. The protocol shall satisfy the next properties except with negligible probability:

- *Termination.* If all honest nodes input some values satisfying Q, then each honest node would output;
- *Agreement.* If two honest nodes output v and v', respectively, then $v = v'$.
- *External-Validity.* If an honest node outputs a value v, then v is valid w.r.t. Q, i.e., $Q(v) = 1$;
- *Quality.* If an honest node outputs v, the probability that v was input by the adversary is at most 1/2.

Note that not all MVBA protocols have the last quality property. For example, a very recent design mentioned in [66] might leave the adversary a chance to always propose the output if without further careful adaption.

Asynchronous common subset (ACS) [23]. In ACS, 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:

- *Validity*. The output set $\mathbf{S}$ of an honest party contains the inputs of at least $n - 2f$ honest parties;
- *Agreement*. The outputs of any two honest parties are same;
- *Termination*. If all honest parties activate the protocol, then all honest parties would output.

CHAPTER 3

DUMBO-MVBA AND ITS APPLICATION TO OPTIMAL ASYNCHRONOUS ATOMIC BROADCAST

This chapter shows how to get the optimal Multi-Valued validated asynchronous Byzantine agreement, and backs it up with solid proofs and analyses. In the end, we explain in detail how to get the optimal atomic broadcast based on Dumbo-MVBA.

3.1 Background

The elegant work of Cachin et al. in 2001 [34] proposed external validity for multi-valued Byzantine agreement (BA) and defined validated asynchronous BA, from which a simple construction of ABC can be achieved. In this multi-valued validated asynchronous BA (MVBA), each party takes a value as input and decides *one* of the values as output, as long as the decided output satisfies the external validity condition. Later, MVBA was used as a core building block to implement a broad array of fault-tolerant protocols beyond ABC [33, 73, 74, 80, 113].

Recently, the renewed attention to multi-valued BA is gathered in the *asynchronous* setting [9, 58, 74, 92], due to an unprecedented demand of deploying asynchronous atomic broadcast (ABC) [39] that is usually instantiated by sequentially executing multi-valued asynchronous BA instances with some fine-tuned validity [34, 49].

3.1.1 Motivation

The first MVBA construction was given in the same paper [34] against computationally-bounded adversaries in the authenticated setting with the random oracle and setup assumptions (e.g., PKI and established threshold cryptosystems). The solution tolerates maximal Byzantine corruptions up to $f < n/3$ and attains expected $O(1)$ running time and $O(n^2)$ messages, but it incurs $O(\ell n^2 + \lambda n^2 + n^3)$ communicated bits, which is large. Here, n

is the number of parties, ℓ represents the bit-length of MVBA input, and λ is the security parameter that captures the bit-length of digital signatures. As such, Cachin et al. raised the open problem of reducing the communication of MVBA protocols (and thus improve their ABC construction) [34], which is rephrased as: *How to asymptotically improve the communication cost of the* MVBA *protocol by an* $O(n)$ *factor?*

After nearly twenty years, in a recent breakthrough of Abraham et al. [9], the n^3 term in the communication complexity was removed, and they achieved optimal $O(n^2)$ word communication, conditioned on each system word can encapsulate a constant number of input values and some small-size strings such as digital signatures. Their result can be directly translated to bit communication as a partial answer to the above question, when the input length ℓ is small (e.g., comparable to λ).

Nevertheless, both of the above MVBA constructions contain the ℓn^2 term in their communication complexities, which was reported in [34] as a major obstacle-ridden factor in a few typical use-cases where the input length ℓ is not that small. For instance, Cachin et al. [34] noticed their ABC construction requires the underlying MVBA's input length ℓ to be at least $O(\lambda n)$, as each MVBA input is a set of $(n - f)$ digitally signed ABC inputs. In this case, the ℓn^2 term becomes the dominating factor. For this reason, it was even considered in [58, 92] that existing MVBA is sub-optimal for constructing ABC due to the large communication. It follows that, despite the recent breakthrough of [9], the question from [34] *remains open* for the moderately large input size $\ell \geq O(\lambda n)$.

3.1.2 Challenges

Let us begin with a very brief tour to revisit the existing MVBA constructions [9, 34]. In the first phase of [34], each party broadcasts its input value to all others using a broadcast protocol. Once receiving sufficient values, each party informs everyone else which values it has received to form a $O(n^2)$ size "matrix". Then a random party $\mathcal{P}_l$ is elected, and an asynchronous binary agreement (ABA) is run by the parties to vote on whether to output v_l depending on if enough parties have already received v_l. The ABA will be repeated until 1

is returned. The recent study [9], instead, expands the conventional design idea of ABA and directly constructs MVBA in the following way: first, multiple rounds of broadcasts are executed by every party to form *commit* proofs. A random party $\mathcal{P}_l$ is elected. If any party already receives a *commit* proof for v_l, it decides to output v_l; and other undecided parties use v_l as input to enter a repetition of the whole procedure. We can see that [9] get rid of the $O(n^3)$ communication as the phase that each party receives a $O(n^2)$ size matrix is removed.

We observe that in the first phase of both [9, 34], every party broadcasts its own input to all parties for checking external validity, which already results in ℓn^2 communicated bits. Note that a MVBA protocol only outputs a single party's input, it is thus unnecessary for *every* party to send its input to all parties. Following the observation, we design Dumbo-MVBA, a novel reduction from MVBA to ABA by using a *dispersal-then-recast* methodology to reduce communication. Instead of letting each party directly send its input to everyone, we let everyone to disperse the coded *fragments* of its input across the network. Later, after the dispersal phase has been completed, the parties could (randomly) choose a dispersed value to collectively recover it. Thanks to the external predicate, all parties can locally check the validity of the recovered value, such that they can consistently decide to output the value, or to repeat random election of another dispersed value to recover.

However, challenges remain due to our multiple efficiency requirements. For example, the number of messages to disperse a value is at most linear, otherwise n dispersals would cost more than quadratic messages and make MVBA not optimal regarding message complexity. The requirement rules out a few related candidates such as asynchronous verifiable information dispersal (AVID) [38, 75] that needs $O(n^2)$ messages to disperse a value. In addition, the protocol must terminate in expected constant time, that means at most a constant number of dispersed values will be recovered on average.

We therefore propose asynchronous provable dispersal broadcast (APDB) for the efficiency purpose, which weakens the agreement of AVID when the sender is corrupted. In this way, we realize a meaningful dispersal protocol with only $O(n)$ messages. We also introduce

two succinct "proofs" in APDB as hinted by the nice work of Abraham et al. [9]. During the dispersal of APDB, two proofs *lock* and *done* could be produced: (i) when any honest party delivers a *lock* proof, enough parties have delivered the coded fragments of the dispersed value, and thus the value can be collectively recovered by all honest parties, and (ii) the *done* proof attests that enough parties deliver *lock*, so all honest parties can activate ABA with input 1 and then decide 1 to jointly recover the dispersed value. To take the most advantage of APDB, we leverage the design in [9] to let the parties exchange their *done* proofs to collectively quit all dispersals, and then borrow the idea in [34] to randomly elect a party and vote via ABA to decide whether to output the elected party's input value (if the value turns to be valid after being recovered). Intuitively, this idea reduces the communication, since (i) each fragment has only $O(\ell/n)$ bits, so n dispersals of ℓ-bit input incur only $O(\ell n)$ bits, (ii) the parties can reconstruct a valid value after expected constant number of ABA and recoveries. See detailed discussions in Section 3.4.

Finally, we present another extension MVBA protocol Dumbo-MVBA$\star$, which is a general self-bootstrap technique to reduce the communication of any existing MVBA. After applying our APDB protocol, we can use small input (i.e., the "proofs" of APDB) to invoke the underlying MVBA to pick the dispersed value to recast, thus reducing the communication of the underlying MVBA. In addition, though Dumbo-MVBA$\star$ is centering around the advanced building block of MVBA instead of the basic module of binary agreement, it can better utilize MVBA to remove the rounds generating the *done* proof in APDB, which further results in a much simpler modular design.

3.2 Related work

Validity conditions. The asynchronous BA problem [23, 30, 40] was studied in diverse flavors, depending on validity conditions.

Strong validity [61, 102] requires that if an honest party outputs v, then v is input of some honest party. This is arguably the strongest notion of validity for multi-valued BA. The sequential execution of BA instances with strong validity gives us an ABC protocol, even in

the asynchronous setting. Unfortunately, implementing strong validity is not easy. In [61], the authors even proved some disappointing bounds of strong validity in the asynchronous setting, which include: (i) the maximal number of corruptions is up to $f < n/(2^\ell + 1)$, and (ii) the optimal running time is $O(2^\ell)$ asynchronous rounds, where ℓ is the input size in bit.

Weak validity [57, 81], only requires that if all honest parties input v, then every honest party outputs v. This is one of most widely adopted validity notions for multi-valued BA. However, it states nothing about output when the honest parties have different inputs. Weak validity is strictly weaker than strong validity [61, 102], except that they coincide in binary BA [36, 86, 99]. Abraham et al. [9] argued: it is not clear how to achieve a simple reduction from ABC to asynchronous multi-valued BA with weak validity; in particular, the sequential execution of multi-valued BA instances with weak validity fails in the asynchronous setting, because non-default output is needed for the liveness [34] or censorship resilience [92].

External validity was proposed by Cachin et al. [34] to circumvent the limits of above validity notions, and it requires the decided output of honest parties to satisfy a globally known predicate. This delicately tuned notion brings a few definitional advantages: (i) compared to strong validity, it is easier to be instantiated, (ii) in contrast with weak validity, ABC is simply reducible to it. For example, Cachin et al. [34] showcased a simple reduction from ABC to MVBA (with using a notion called asynchronous common subset, i.e., ACS, as a bridge). This succinct construction sequentially executes the ACS instances, each of which allows every party to propose an input value and then solicits $n - f$ input values (from distinct parties) to output. The work also instantiates ACS due to a reduction to MVBA by centering around a specific external validity condition, namely, input/output must be a set containing $2f + 1$ valid message-signature pairs generated by distinct parties, where each signed message is an ABC input. Although their reduction from ABC to MVBA is arguably simple, the communication cost (per delivered bit) in their ABC was cubic (and is still amortizedly quadratic even if using the recent technique of batching in [92]), mainly because (i) the reduction to ABC requires each MVBA input consisting of $O(n)$

ABC inputs and $O(n)$ digital signatures, and (ii) the communication cost of the underlying MVBA module contains a quadratic term factored by the MVBA's input length.

BA **extension protocols**. In the asynchronous setting, there exist a few nice extension protocols that can invoke Byzantine agreement with using short input to accommodate large multi-valued input [62, 101, 106]. To the best of our knowledge, all the existing extension protocols focus on weak validity, and their techniques cannot be directly borrowed to handle external validity. The challenge in our extension MVBA protocol Dumbo-MVBA$\star$ stems from that the externally valid output can come from one corrupt party. More specifically, in the extension protocols for weak validity, it has to decides an output only when all honest parties share the same input value, which can be ensured through a simple dispersal-recast technique [101] since sufficient honest parties already share the same input; while in our case of MVBA for external validity, when the decided output is from a malicious party, there are no sufficient honest parties share the same value as input to assist the recovery, so we have to ensure the value is indeed correctly dispersed and becomes recoverable (which is realized via the recastability in our APDB) by forcing each of them to attach an extra short proof.

Roadmap to asynchronous atomic broadcast. To make ABC practical, most existing protocols [34, 49, 58, 74, 92] are instantiated via the component of ACS [23]. ACS allows every party to take a value as input, and decides a common subset as output to include sufficient input values from distinct parties. ABC can be instantiated by simply executing ACS instances sequentially [34, 37], or by some more involved techniques batching threshold encryption [92].

There are two main methods to construct ACS protocols: one is initiated by Cachin et al. [34] to reduce ACS to MVBA. The other method, initiated by Ben-Or et al. [23] and recently improved by Miller et al. in HoneyBadgerBFT (HBBFT) [92], builds ACS using n asynchronous binary agreements (ABAs) directly. During the past years, the former approach (i.e., building ACS from MVBA) was considered as sub-optimal to instantiate

ABC in literature [58, 92], because of the large communication complexity of existing MVBA protocols.

In a very recent work, Dumbo BFT [74], proposes a novel reduction from ACS to MVBA, which results in an ACS protocol that attains only constant running time compared to that in [58, 92] depending on the number of parties n while remaining the same communication and message complexities remain. More importantly, this is achieved despite invoking existing MVBA protocols with seemingly large communication complexity, e.g., the first MVBA proposed by Cachin et al. two decades ago. In this paper, we directly reduce the communication complexity of MVBA protocols for a factor of n, which corresponds to another evidence that the earlier belief [58, 92] that MVBA is sub-optimal could be too pessimistic to be true. These two results together show that MVBA is still be the right way to construct efficient ACS as the bridge to practical ABC!

For sake of completeness, we also present how to build practical ABC around the communication-optimal MVBA protocols. The approach uses our MVBA protocols as the underlying building blocks to improve Cachin et al.'s ACS construction, so an $O(n)$-factor improvement of communication complexity in ABC can be immediately obtained; then, the batching technique recently invented by Miller et al. in [92] is borrowed to further optimize the communication usage, so in some extreme scenarios (e.g., each party's network bandwidth is sufficient for batching), the communication cost can be further reduced by another $O(n)$-factor.

3.3 Problem Formulation

3.3.1 System model

We use the standard notion [9, 34] to model the system consisting of n parties and an adversary in the *authenticated setting*.

Established identities & trusted setup. There are n designated parties denoted by $\{\mathcal{P}_i\}_{i\in[n]}$, where $[n]$ is short for $\{1,\ldots,n\}$ through the paper. Moreover, we consider the trusted setup of threshold cryptosystems, namely, before the start of the protocol, each party has gotten

its own secret key share and the public keys as internal states. For presentation simplicity, we consider this trusted setup for granted, while in practice it can be done via a trusted dealer or distributed key generation protocols [29, 76, 83].

Adaptive Byzantine corruption. The adversary $\mathcal{A}$ can adaptively corrupt any party at any time during the course of protocol execution, until $\mathcal{A}$ already controls f parties (e.g., $3f + 1 = n$). If a party $\mathcal{P}_i$ was not corrupted by $\mathcal{A}$ at some stage of the protocol, it followed the protocol and kept all internal states secret against $\mathcal{A}$, and we say it is *so-far-uncorrupted*. Once a party $\mathcal{P}_i$ is corrupted by $\mathcal{A}$, it leaks all internal states to $\mathcal{A}$ and remains fully controlled by $\mathcal{A}$ to arbitrarily misbehave. By convention, the corrupted party is also called *Byzantine fault*. If and only if a party is not corrupted through the entire execution, we say it is *honest*.

Computation model. Following standard cryptographic practices [34, 36], we let the n parties and the adversary $\mathcal{A}$ to be probabilistic polynomial-time interactive Turing machines (ITMs). A party $\mathcal{P}_i$ is an ITM defined by the protocol: it is activated upon receiving an incoming message to carry out some computations, update its states, possibly generate some outgoing messages, and wait for the next activation. $\mathcal{A}$ is a probabilistic ITM that runs in polynomial time (in the number of message bits generated by honest parties). Moreover, we explicitly require the message bits generated by honest parties to be probabilistic uniformly bounded by a polynomial in the security parameter λ, which was formulated as *efficiency* in [9, 34] to rule out infinite protocol executions and thus restrict the run time of the adversary through the entire protocol. Same to [34] and [9], all system parameters (e.g., n) are bounded by polynomials in λ.

Asynchronous network. Any two parties are connected via an asynchronous *reliable authenticated* point-to-point channel. When a party $\mathcal{P}_i$ attempts to send a message to another party $\mathcal{P}_j$, the adversary $\mathcal{A}$ is firstly notified about the message; then, $\mathcal{A}$ fully determines when $\mathcal{P}_j$ receives the message, but cannot drop or modify this message if both $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest. The network model also allows the adaptive adversary $\mathcal{A}$ to perform

"after-the-fact removal", that is, when $\mathcal{A}$ is notified about some messages sent from a *so-far-uncorrupted* party $\mathcal{P}_i$, it can delay these messages until it corrupts $\mathcal{P}_i$ to drop them.

3.3.2 Security goal

We review hereunder the definition of (multi-valued) validated asynchronous Byzantine agreement (MVBA) due to [9, 34].

Definition 2. *In an* MVBA *protocol with an external* Predicate : $\{0,1\}^\ell \to \{true, false\}$, *the parties take values satisfying* Predicate *as inputs and aim to output a common value satisfying* Predicate. *The* MVBA *protocol guarantees the following properties, except with negligible probability, for any identification* id, *in the asynchronous authenticated model:*

1. **Termination.** *If every honest party $\mathcal{P}_i$ is activated on identification* id, *with taking as input a value v_i s.t.* Predicate$(v_i) = true$, *then every honest party outputs a value v for* id.
2. **External-Validity.** *If an honest party outputs a value v for* id, *then* Predicate$(v) = true$.
3. **Agreement.** *If any two honest parties output v and v' for* id *respectively, then $v = v'$.*
4. **Quality.** *If an honest party outputs v for* id, *the probability that v was proposed by the adversary is at most 1/2.*

We make the following remarks about the above definition:

1. *Input length.* We focus on the general case that the input length ℓ can be a function in n. We emphasize that it captures many realistic scenarios. One remarkable example is to build ABC around MVBA as in [34] where the length of each MVBA input is at least $O(\lambda n)$.
2. *External-validity* is a fine-grained validity requirement of BA. In particular, it requires the common output of the honest parties to satisfy a pre-specified global predicate function.

3. *Quality* was proposed by Abraham et al. in [9], which not only rules out trivial solutions w.r.t. some trivial predicates (e.g., output a known valid value) but also captures "fairness" to prevent the adversary from fully controlling the output.

3.4 APDB: Asynchronous Provable Dispersal Broadcast

The dominating $O(\ell n^2)$ term in the communication complexity of existing MVBA protocols [9, 34] is because every party broadcasts its own input *all* other parties. This turns out to be unnecessary, as in the MVBA protocol, only one single party's input is decided as output. To remedy the needless communication overhead in MVBA, we introduce a new *dispersal-then-recast* methodology, through which each party $\mathcal{P}_i$ only has to spread the coded *fragments* of its input v_i to every other party instead of its entire input.

This section introduces the core building block, namely, the asynchronous provable dispersal broadcast (APDB), to instantiate the *dispersal-then-recast* idea. The notion is carefully tailored to be efficiently implementable. Especially, in contrast to related AVID protocols [38, 75], APDB can disperse a value at a cost of linear messages instead of $O(n^2)$, as a reflection of following trade-offs:

- The APDB notion weakens AVID, so upon that a party outputs a coded fragment in the dispersal instance of APDB, there is no guarantee that other parties will output the consistent fragments. Thus, it could be not enough to recover the dispersed value by only $f + 1$ honest parties, as these parties might receive (probably inconsistent) fragments.

- To compensate the above weakenings, we let the sender to spread the coded fragments of its input along with a succinct vector commitment of all these fragments, and then produce two succinct "proofs" *lock* and *done*. The "proofs" facilitate: (i) the *lock* proof ensures that $2f + 1$ parties receive some fragments that are committed in the same vector commitment, so the honest parties can either recover the same value, or output $\perp$ (that means the committed fragments are inconsistent); (ii) the *done* proof ensures that $2f + 1$ parties deliver valid *lock*s, thus allowing the parties to reach a

common decision, e.g., via a (biased) binary BA [34], to all agree to jointly recover the dispersed value, which makes the value deemed to be recoverable.

In this way, the overall communication of dispersing a value can be brought down to minimum as the size of each fragment is only $O(\ell/n)$ where ℓ is the length of input v. Moreover, this well-tuned notion can be easily implemented in light of [9] and costs only linear messages. These efficiencies are needed to achieve the optimal communication and message complexities for MVBA.

Defining asynchronous provable dispersal broadcast. Formally, the syntax and properties of a APDB protocol are defined as follows.

Definition 3. *An* APDB *protocol with a designated sender $\mathcal{P}_s$ is equipped with a pair of predicates* (ValidateLock, ValidateDone) *and consists of a provable dispersal subprotocol (*PD*) and a recast subprotocol (*RC*) as follows:*

- PD **subprotocol**. *In the* PD *subprotocol (with identifier* ID*) among n parties, a designated sender $\mathcal{P}_s$ inputs a value $v \in \{0,1\}^\ell$, and aims to split v into n encoded fragments and disperses each fragment to the corresponding party. During the* PD *subprotocol with identifier* ID*, each party is allowed to invoke an abandon*(ID) *function. After* PD *terminates, each party shall output two strings store and lock, and the sender shall output an additional string done.*

  Note that the lock and done strings are said to be valid for the identifier ID*, if and only if* ValidateLock(ID, *lock*) $= 1$ *and* ValidateDone(ID, *done*) $= 1$*, respectively.*

- RC **subprotocol**. *In the* RC *subprotocol (with identifier* ID*), all honest parties take the output of the* PD *subprotocol (with the same* ID*) as input, and aim to output the value v that was dispersed in the* RC *subprotocol. Once* RC *is completed, the parties output a common value in $\{0,1\}^\ell \cup \perp$.*

An APDB *protocol (*PD, RC*) with identifier* ID *satisfies the following properties in the asynchronous authenticated setting (c.f. Section 3.3), except with negligible probability:*

- **Termination**. *If the sender $\mathcal{P}_s$ is honest and all honest parties activate* PD[ID] *without invoking abandon*(ID)*, then each honest party would output store and valid lock for* ID*; additionally, the sender $\mathcal{P}_s$ outputs valid done for* ID.

- **Recast-ability**. *If all honest parties invoke* RC[ID] *with inputting the output of* PD[ID] *and at least one honest party inputs a valid lock, then: (i) all honest parties recover a common value; (ii) if the sender dispersed v in* PD[ID] *and has not been corrupted before at least one party delivers valid lock, then all honest parties recover v in* RC[ID].

  Intuitively, the recast-ability *captures that the valid lock is a "proof" attesting that the input value dispersed via* PD[ID] *can be consistently recovered by* all *parties through collectively running the corresponding* RC[ID] *instance.*

- **Provability**. *If the sender of* PD[ID] *produces valid done, then at least $f + 1$ honest parties output valid lock.*

  Intuitively, the provability *indicates that done is a "completeness proof" attesting that at least $f + 1$ honest parties output valid locks, such that the parties can exchange locks and then vote via* ABA *to reach an agreement that the dispersed value is deemed recoverable.*

- **Abandon-ability**. *If every party (and the adversary) cannot produce valid lock for* ID *and $f + 1$ honest parties invoke abandon*(ID)*, no party would deliver valid lock for* ID.

3.4.1 Overview of the APDB protocol

For the PD subprotocol with identifier ID, it has a simple structure of four one-to-all or all-to-one rounds: sender $\xrightarrow{\text{Store}}$ parties $\xrightarrow{\text{Stored}}$ sender $\xrightarrow{\text{Lock}}$ parties $\xrightarrow{\text{Locked}}$ sender. Through a Store message, every party $\mathcal{P}_i$ receives $store := \langle \mathsf{vc}, m_i, i, \pi_i \rangle$, where m_i is an encoded fragment of the sender's input, vc is a (deterministic) commitment of the vector of all fragments, and π_i attests m_i's inclusion in vc at the i-th position; then, through Stored messages, the

parties would give the sender "partial" signatures for the string $\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc} \rangle$; next, the sender combines $2f + 1$ valid "partial" signatures, and sends every party the combined "full" signature σ_1 for the string $\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc} \rangle$ via Locked messages, so each party can deliver $lock := \langle \mathsf{vc}, \sigma_1 \rangle$; finally, each party sends a "partial" signature for the string $\langle \textsc{Locked}, \mathsf{ID}, \mathsf{vc} \rangle$, such that the sender can again combine the "partial" signatures to produce a valid "full" signature σ_2 for the string $\langle \textsc{Locked}, \mathsf{ID}, \mathsf{vc} \rangle$, which allows the sender to deliver $done := \langle \mathsf{vc}, \sigma_2 \rangle$.

For the RC subprotocol, it has only one-round structure, as each party only has to take some output of PD subprotocol as input (i.e., *lock* and *store*), and multicasts these inputs to all parties. As long as an honest party inputs a valid *lock*, there are at least $f + 1$ honest parties deliver valid *store*s that are bound to the vector commitment vc included in *lock*, so all parties can eventually reconstruct the dispersed value that was committed in the commitment vc.

Algorithm 1 Validation func of APDB protocol, with identifier ID

▷ ValidateStore verifies *store* commits a fragment m_i in vc at i-th position
function ValidateStore(i', *store*):
1: parse *store* as $\langle \mathsf{vc}, i, m_i, \pi_i \rangle$
2: return VerifyOpen($\mathsf{vc}, m_i, i, \pi_i$) $\wedge\ i = i'$

▷ ValidateLock validates *lock* contains a vc signed by $2f + 1$ parties
function ValidateLock(ID, *lock*):
3: parse *lock* as $\langle \mathsf{vc}, \sigma_1 \rangle$
4: return $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc} \rangle, \sigma_1)$

▷ ValidateDone validates *done* to attest $2f + 1$ parties receive valid *lock*
function ValidateDone(ID, *done*):
5: parse *done* as $\langle \mathsf{vc}, \sigma_2 \rangle$
6: return $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Locked}, \mathsf{ID}, \mathsf{vc} \rangle, \sigma_2)$

3.4.2 Details of the APDB protocol

As illustrated in Algorithm 1, the APDB protocol is designed with a few functions called as ValidateStore, ValidateLock and ValidateDone to validate *done*, *lock* and *store*, respectively. ValidateStore is to check the *store* received by the party $\mathcal{P}_i$ includes a fragment m_i that is committed in a vector commitment vc at the i-th position, ValidateLock

validates *lock* to verify that $2f + 1$ parties (i.e., at least $f + 1$ honest parties) receive the fragments that are correctly committed in the same vector commitment vc, and ValidateDone validates *done* to verify that $2f + 1$ parties (i.e., at least $f + 1$ honest parties) have delivered valid *lock*s (that contain the same vc).

PD *subprotocol*. The details of the PD subprotocol are shown in Algorithm 2. In brief, a PD instance with identifier ID (i.e., PD[ID]) allows a designated sender $\mathcal{P}_s$ to disperse a value v as follows:

1. *Store-then-Stored* (line 1-6, 13-15, 19-23). When the sender $\mathcal{P}_s$ receives an input value v to disperse, it encodes v to generate a vector of coded fragments $m = (m_1, \ldots, m_n)$ by an $(f + 1, n)$-erasure code; then, $\mathcal{P}_s$ commits m in a vector commitment vc. Then $\mathcal{P}_s$ sends *store* including the commitment vc, the i-th coded fragment m_i and the commitment opening π_i to each party $\mathcal{P}_i$ by Store messages. Upon receiving (Store, ID, *store*) from the sender, $\mathcal{P}_i$ verifies whether *store* is valid. If that is the case, $\mathcal{P}_i$ delivers *store* and sends a $(2f + 1, n)$-partial signature $\rho_{1,i}$ for $\langle$Stored, ID, vc$\rangle$ back to the sender through a Stored message.

2. *Lock-then-Locked* (line 7-9, 16-18, 24-28). Upon receiving $2f + 1$ valid Stored messages from distinct parties, the sender $\mathcal{P}_s$ produces a full signature σ_1 for the string $\langle$Stored, ID, vc$\rangle$. Then, $\mathcal{P}_s$ sends *lock* including vc and σ_1 to all parties through Lock messages. Upon receiving Lock message, $\mathcal{P}_i$ verifies whether σ_1 is deemed as a valid full signature. If that is the case, $\mathcal{P}_i$ delivers *lock* $= \langle$vc, $\sigma_1\rangle$, and sends a $(2f + 1, n)$-partial signature $\rho_{2,i}$ for the string $\langle$Locked, ID, vc$\rangle$ back to the sender through a Locked message.

3. *Done* (line 10-12). Once the sender $\mathcal{P}_s$ receives $2f + 1$ valid Locked messages from distinct parties, it produces a full signature σ_2 for $\langle$Locked, ID, vc$\rangle$. Then $\mathcal{P}_s$ outputs the completeness proof *done* $= \langle$vc, $\sigma_2\rangle$ and terminates the dispersal.

4. *Abandon* (line 29). A party can invoke *abandon*(ID) to explicitly stop its participation in this dispersal instance with identification ID. In particular, if $f + 1$ honest parties

Algorithm 2 PD subprotocol, with identifier ID and sender $\mathcal{P}_s$

let $S_1 \leftarrow \{\}$, $S_2 \leftarrow \{\}$, $stop \leftarrow 0$

/* Protocol for the sender $\mathcal{P}_s$ */

1: **upon** receiving an input value v **do**
2: $m \leftarrow \mathsf{Enc}(v)$, where v is parsed as a $f+1$ vector and m is a n vector
3: $\mathsf{vc} \leftarrow \mathsf{VCom}(m)$
4: **for** each $j \in [n]$ **do**
5: $\pi_j \leftarrow \mathsf{Open}(\mathsf{vc}, m_j, j)$
6: **let** $store := \langle \mathsf{vc}, m_j, j, \pi_j \rangle$
7: **send** (STORE, ID, $store$) to $\mathcal{P}_j$ ▷ send *store*
8: **wait** until $|S_1| = 2f+1$
9: $\sigma_1 \leftarrow \mathsf{Combine}_{(2f+1)}(\langle \text{STORED}, \mathsf{ID}, \mathsf{vc} \rangle, S_1)$
10: **let** $lock := \langle \mathsf{vc}, \sigma_1 \rangle$
11: **multicast** (LOCK, ID, $lock$) to all parties ▷ multicast *lock* proof
12: **wait** until $|S_2| = 2f+1$
13: $\sigma_2 \leftarrow \mathsf{Combine}_{(2f+1)}(\langle \text{LOCKED}, \mathsf{ID}, \mathsf{vc} \rangle, S_2)$
14: **let** $done := \langle \mathsf{vc}, \sigma_2 \rangle$ and **deliver** $done$ ▷ produce *done* proof
15: **upon** receiving (STORED, ID, $\rho_{1,j}$) from $\mathcal{P}_j$ for the first time **do**
16: **if** $\mathsf{VerifyShare}_{(2f+1)}(\langle \text{STORED}, \mathsf{ID}, \mathsf{vc} \rangle, (j, \rho_{1,j})) = 1$ and $stop = 0$ **then**
17: $S_1 \leftarrow S_1 \cup (j, \rho_{1,j})$
18: **upon** receiving (LOCKED, ID, $\rho_{2,j}$) from $\mathcal{P}_j$ for the first time **do**
19: **if** $\mathsf{VerifyShare}_{(2f+1)}(\langle \text{LOCKED}, \mathsf{ID}, \mathsf{vc} \rangle, (j, \rho_{2,j})) = 1$ and $stop = 0$ **then**
20: $S_2 \leftarrow S_2 \cup (j, \rho_{2,j})$

/* Protocol for each party $\mathcal{P}_i$ */

21: **upon** receiving (STORE, ID, $store$) from sender $\mathcal{P}_s$ for the first time **do**
22: **if** $\mathsf{ValidateStore}(i, store) = 1$ and $stop = 0$ **then** ▷ receive *store*
23: **deliver** $store$ and parse it as $\langle \mathsf{vc}, i, m_i, \pi_i \rangle$
24: $\rho_{1,i} \leftarrow \mathsf{SignShare}_{(2f+1)}(sk_i, \langle \text{STORED}, \mathsf{ID}, \mathsf{vc} \rangle)$
25: **send** (STORED, ID, $\rho_{1,i}$) to $\mathcal{P}_s$
26: **upon** receiving (LOCK, ID, $lock$) from sender $\mathcal{P}_s$ for the first time **do**
27: **if** $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$ and $stop = 0$ **then**
▷ receive *lock*
28: **deliver** $lock$ and parse it as $\langle \mathsf{vc}, \sigma_1 \rangle$
29: $\rho_{2,i} \leftarrow \mathsf{SignShare}_{(2f+1)}(sk_i, \langle \text{LOCKED}, \mathsf{ID}, \mathsf{vc} \rangle)$
30: **send** (LOCKED, ID, $\rho_{2,i}$) to $\mathcal{P}_s$

procedure *abandon*(ID):
31: $stop \leftarrow 1$

Algorithm 3 RC subprotocol with identifier ID, for each party $\mathcal{P}_i$

let $C \leftarrow [\,]$

1: **upon** receiving input $(store, lock)$ **do** ▷ multicast *lock* and/or *store*
2: **if** $lock \neq \emptyset$ **then**
3: **multicast** (RcLock, ID, *lock*) to all
4: **if** $store \neq \emptyset$ **then**
5: **multicast** (RcStore, ID, *store*) to all
6: **upon** receiving (RcLock, ID, *lock*) **do**
▷ assert: only one valid *lock* for each ID (c.f. Lemma 3)
7: **if** ValidateLock(ID, *lock*) = 1 **then**
8: **multicast** (RcLock, ID, *lock*) to all, if was not sent before
9: parse *lock* as $\langle \mathsf{vc}, \sigma_1 \rangle$
10: **wait** until $|C[\mathsf{vc}]| = f + 1$
11: $v \leftarrow \mathsf{Dec}(C[\mathsf{vc}])$
12: **if** VCom(Enc(v)) = vc **then** ▷ deliver the dispersed value
13: **return** v
14: **else return** $\perp$
15: **upon** receiving (RcStore, ID, *store*) from $\mathcal{P}_j$ for the first time **do**
16: **if** ValidateStore(j, *store*) = 1 **then**
17: parse *store* as $\langle \mathsf{vc}, m_j, j, \pi_j \rangle$
▷ record fragments committed to each vc
18: $C[\mathsf{vc}] \leftarrow C[\mathsf{vc}] \cup (j, m_j)$
19: **else discard** the invalid message

invoke *abandon*(ID), the adversary can no longer corrupt the sender of PD[ID] to disperse anything across the network.

RC *subprotocol*. The construction of the RC subprotocol is shown in Algorithm 3. The input of RC subprotocol consists of *lock* and *store*, which were probably delivered during the PD subprotocol. In brief, the execution of a RC instance with identification ID is as:

1. *Recast* (line 1-5). If the party $\mathcal{P}_i$ inputs *lock* and/or *store*, it multicasts them to all parties.

2. *Deliver* (line 6-18). If the party $\mathcal{P}_i$ receives a valid *lock* message, it waits for $f + 1$ valid *store*s bound to this *lock*, such that $\mathcal{P}_i$ can reconstruct a value v (or a special symbol $\perp$).

3.4.3 Analyses of the APDB protocol

Here we present the detailed proofs along with the complexity analyses for APDB protocol.

Security intuition. The tuple of protocols in Algorithm 2 and 3 realize APDB among n parties against the adaptive adversary controlling up to $f \leq \lfloor \frac{n-1}{3} \rfloor$ parties, given (i) $(f+1, n)$-erasure code, (ii) deterministic n-vector commitment with the position-binding property, and (iii) established $(2f+1, n)$-threshold signature with adaptive security. The high-level intuition is: (i) if any honest party outputs valid *lock*, then at least $f+1$ honest parties receives the code fragments committed in the same vector commitment, and the position-binding property ensures that the honest parties can collectively recover a common value (or the common $\perp$) from these committed fragments; (ii) whenever any party can produce a valid *done*, it attests that $2f+1$ (namely, at least $f+1$ honest) parties have indeed received valid *lock*s.

The proofs of APDB. Now we prove the Algorithm 2 and 3 would satisfy the properties of APDB as defined in Definition 3 with all but negligible probability.

Corollary 3.4.1. *Considering a n-vector commitment scheme* (VCom, Open, VerifyOpen) *and a* (k, n)*-erasure coding scheme* (Enc, Dec)*, we have the following observations:*

Correctness. For any $S \subset [n]$ *that* $|S| = k$, $\Pr[\mathsf{Dec}(\{(i, m_i)\}_{j \in S}) = v \wedge \mathsf{VCom}(\mathsf{Enc}(v)) = \mathsf{vc} \mid (m_1, \cdots, m_n) \leftarrow \mathsf{Enc}(v) \wedge \mathsf{vc} \leftarrow \mathsf{VCom}((m_1, \cdots, m_n))] = 1$. *The property reveals: if a vector of the coded fragments of a value v are committed to a vector commitment* vc*, then any k-subset of the fragments can recover the original value v, whose encoded fragments can be used to produce the same commitment* vc*. This is true, because of the correctness of erasure coding and the determinism of* VCom*.*

Security. For any $S_1 \subset [n]$ *and* $S_2 \subset [n]$ *that* $|S_1| = |S_2| = k$ *and* $S_1 \neq S_2$*, it is computationally infeasible for any P.P.T. adversary (on input the security parameter* λ *and all other public system parameters) to produce vc,* $\{(i, m_i, \pi_i)\}_{i \in S_1}$ *and* $\{(j, m_j, \pi_j)\}_{j \in S_2}$ *where* $\forall i \in S_1, \mathsf{VerifyOpen}(\mathsf{vc}, m_i, i, \pi_i) = 1$ *and* $\forall j \in S_2, \mathsf{VerifyOpen}(\mathsf{vc}, m_j, j, \pi_j) = 1$*, such that: (i)* $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(i, m_i)\}_{i \in S_1}))) = \mathsf{vc} \wedge \mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(j, m_j)\}_{j \in S_2}))) = \mathsf{vc}$

$\wedge$ $\mathsf{Dec}(\{(i, m_i)\}_{i\in S_1}) \neq \mathsf{Dec}(\{(j, m_j)\}_{j\in S_2})$*, or (ii)* $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(i, m_i)\}_{i\in S_1}))) = \mathsf{vc} \wedge \mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(j, m_j)\}_{j\in S_2}))) \neq \mathsf{vc}$*. The property indicates whenever the (probably malicious generated) coded fragments are committed to* vc*, any two k-subsets of these committed fragments must: either consistently recover a common value that can re-produce the same commitment* vc*, or recover some values that simultaneously re-produce some commitments different from* vc.

Lemma 1. Termination. *If the sender $\mathcal{P}_s$ is honest and all honest parties activate* PD[ID] *without invoking abandon*(ID)*, then each honest party would output store and valid lock for* ID *s.t.* $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$*; additionally, the sender $\mathcal{P}_s$ outputs valid done for* ID.

Proof. In case all honest parties activate PD[ID] without abandoning, all honest parties will follow the PD protocol specified by the pseudocode shown in Algorithm 2. In addition, since the sender $\mathcal{P}_s$ is honest, it also follows the protocol.

Due to the PD protocol, the honest sender firstly sends (Store, ID, *store*) to $\mathcal{P}_i$, where the Store message satisfies $\mathsf{ValidateStore}(i, store) = 1$; after receiving the valid Store message, all honest parties will send Stored messages back to the sender. When $\mathcal{P}_s$ receiving $2f + 1$ (the number of honest parties at least is $2f + 1$) valid Stored messages, $\mathcal{P}_s$ will combine these messages to generate valid signature σ_1 by $\mathsf{Combine}_{(2f+1)}$ algorithm and then obtain a valid *lock*. After that, $\mathcal{P}_s$ will send (Lock, ID, *lock*) to all, where the Lock message satisfies $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$.

Next, after receiving the valid Lock message, all honest parties will send Locked message back to the sender. When receiving $2f + 1$ valid Locked message, the sender can generate valid signature σ_2 by $\mathsf{Combine}_{(2f+1)}$ algorithm. Hence, the honest sender $\mathcal{P}_s$ can outputs a completeness proof $done = \langle \mathsf{vc}, \sigma_2 \rangle$, s.t. $\mathsf{ValidateDone}(\mathsf{ID}, done) = 1$. □

Lemma 2. Provability. *If the sender of* PD[ID] *can produce done s.t.* $\mathsf{ValidateDone}(\mathsf{ID}, done) = 1$*, then at least $f + 1$ honest parties output lock s.t.* $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$.

Proof. ValidateDone(ID, *done*) = 1 is equivalent to that $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Locked}, \mathsf{ID}, \mathsf{vc}\rangle, \sigma_2) = 1$ due to Algorithm 1, which means that without overwhelming probability, the sender does receive at least $2f + 1$ Locked message from distinct $\mathcal{P}_j$ to generate σ_2 (otherwise the threshold signature can be forged). With all but negligible probability, at least $f + 1$ honest parties send Locked messages to the sender with attaching their "partial" signatures for $\langle \textsc{Locked}, \mathsf{ID}, \mathsf{vc}\rangle$. From Algorithm 2, we know the honest parties sends their "partial" signatures for $\langle \textsc{Locked}, \mathsf{ID}, \mathsf{vc}\rangle$ to the sender, iff they deliver valid *lock* which satisfies ValidateLock(ID, *lock*) = 1. So this lemma holds with overwhelming probability, otherwise it would break the unforgeability of the underlying threshold signature. □

Lemma 3. *If two parties $\mathcal{P}_i$ and $\mathcal{P}_j$ deliver lock and lock′ in* RC[ID] *and* ValidateLock(ID, *lock*) = 1 ∧ ValidateLock(ID, *lock′*) = 1*, then lock = lock′.*

Proof. When ValidateLock(ID, *lock*) = 1, we can assert that $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc}\rangle, \sigma_1) = 1$ due to Algorithm 1. According to the Algorithm 2, σ_1 was generated by combining $2f + 1$ distinct parties' partial signatures for $\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc}\rangle$. Therefore, there have at least $f + 1$ honest parties produced a share signature for (Stored, ID, vc).

However, if ValidateLock(ID, *lock′*) = 1, $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc}'\rangle, \sigma'_1) = 1$ also holds, which means that there are at least $f + 1$ honest parties that also produce a share signature for (Stored, ID, vc′). Hence, at least one honest party produce two different Stored message if *lock* ≠ *lock′*. However, every honest parties compute the share signature for Stored message at most once for a given identifier ID. Hence, we have *lock* = *lock′* with overwhelming probability; otherwise, the unforgeability of the underlying threshold signature would be broken. □

Lemma 4. Recast-ability. *If all honest parties invoke* RC[ID] *with inputting the output of* PD[ID] *and at least one honest party inputs a valid lock, then: (i) all honest parties recover a common value; (ii) if the sender dispersed v in* PD[ID] *and has not been corrupted before at least one party delivers valid lock, then all honest parties recover v in* RC[ID].

Proof. To prove the conclusion (i) of the Lemma, we would prove the following two statements: first, all honest parties can output a value; second, the output of any two honest parties would be same.

Part 1: Since at least one honest party delivers *lock* satisfying $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$, according to the Algorithm 3, it will multicast (RcLock, ID, *lock*) to all. Hence, all honest parties can receive the valid *lock*.

Note whenever a valid $lock := \langle \mathsf{vc}, \sigma_1 \rangle$ can be produced, there is a valid threshold signature σ_1 was generated by combining $2f+1$ distinct parties' partial signatures for (Store, ID, vc), due to Algorithm 2. Also notice that an honest party partially signs (Store, ID, vc), iff it delivers valid *store* that is bound to the commitment vc. So there are at least $f + 1$ honest parties deliver the valid *store* that are committed to the same commitment string vc.

Thus there have at least $f + 1$ honest parties $\mathcal{P}_i$ will multicast valid (RcStore, ID, *store*) message to all, due to Algorithm 3. For each honest party, they can eventually receive a valid valid RcLock message and $f + 1$ valid RcStore messages, that are corresponding to the same vc. So all parties will always attempt the decode the received fragments in the RcStore messages to eventually recover some value.

Part 2: From the Lemma 3, all honest parties would receive valid RcLock messages with the same $lock := \langle \mathsf{vc}, \sigma_1 \rangle$. Therefore, each honest party can receive $f + 1$ valid RcStore messages, which contain $f+1$ fragments that are committed to the same vector commitment vc. Due to Corollary 3.4.1, either every honest party $\mathcal{P}_i$ have $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(C[\mathsf{vc}]))) = \mathsf{vc}$ or every honest party $\mathcal{P}_i$ have $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(C[\mathsf{vc}]))) \neq \mathsf{vc}$. Therefore, either all honest parties return a common value $\mathsf{Dec}(C[\mathsf{vc}])$ in $\{0,1\}^{\ell}$, or they return a special symbol $\perp$.

The conclusion (i) of the Lemma holds immediately by following Part 1 and Part 2.

For the conclusion (ii) of the Lemma, if the sender $\mathcal{P}_s$ has not been corrupted (so-far-uncorrupted) before at least one party delivers valid *lock* and passed the value v into $\mathsf{PD}[\mathsf{ID}]$ as input, the sender would at least follow the protocol to send Store messages for

dispersing v. Moreover, when the so-far-uncorrupted $\mathcal{P}_s$ delivers valid *lock*, at least $f+1$ honest parties already receive the STORE messages for dispersing v, so the adversary can no longer corrupts $\mathcal{P}_s$ to disperse a value v' different from v, as it cannot produce valid *lock* or valid *done* for v'. From the proving of conclusion (i), we know all parties would recover the value $\mathsf{Dec}(C[\mathsf{vc}])$, which must be v due to the properties of erasure code and vector commitment. □

Lemma 5. Abandon-ability. *If every party (and the adversary) cannot produce valid lock for* ID *and* $f+1$ *honest parties invoke abandon*(ID)*, no party would deliver valid lock for* ID.

Proof. From Algorithm 2, we know it needs $2f+1$ valid STORED messages to produce a valid $lock := \langle \mathsf{vc}, \sigma_1 \rangle$. Since any parties (including the adversary) has not yet produced a valid *lock* and $f+1$ honest parties invoke *abandon*(ID), there are at most $2f$ parties are participating in the PD[ID] instance. So there are at most $2f$ valid STORED messages, which are computationally infeasible for any party to produce a valid *lock*; otherwise, the unforgeability of underlying threshold signature would not hold. □

Theorem 1. *The tuple of protocols described by Algorithms 2 and 3 solves asynchronous provable dispersal broadcast (*APDB*) among n parties against an adaptive adversary controlling* $f < n/3$ *parties, given (i)* $(f+1, n)$*-erasure code, (ii) n-vector commitment scheme, and (iii) established non-interactive* $(2f+1, n)$*-threshold signature with adaptive security.*

Proof. Lemma 1, 2, 4 and 5 complete the proof. □

The complexity analysis of APDB. Through this paper, we consider ℓ is the input length and λ is cryptographic security parameter (the length of signature, vector commitment, and openness proof for commitment), then:

- PD **complexities:** According to the process of Algorithm 2, the PD subprotocol has 4 one-to-all (or all-to-one) rounds. Hence, the total number of messages sent by honest

parties is at most $4n$, which attains $O(n)$ messages complexity and $O(1)$ running time. Besides, the maximal size of messages is $O(\ell/n + \lambda)$, so the communication complexity of PD is $O(\ell + n\lambda)$.

- RC **complexities:** According to the process of Algorithm 3, the message exchanges appear in two places. First, all parties multicast the RcLock messages to all, so the first parts' messages complexity is $O(n^2)$; second, all parties multicast the RcStore messages to all, thus the second parts incurring $O(n^2)$ messages. Hence, the RC incurs $O(n^2)$ messages complexity and constant running time. Besides, each RcLock message is sized to $O(\lambda)$-bit, and the size of each RcStore message is $O(\ell/n + \lambda)$-bit, so the communication complexity of RC is $O(n^2\lambda) + O(n\ell + n^2\lambda) = O(n\ell + n^2\lambda)$.

3.5 Dumbo-MVBA: An Optimal MVBA Protocol

We now apply our *dispersal-then-recast* methodology to design the optimal MVBA protocol Dumbo-MVBA, using APDB and ABA. It is secure against adaptively corrupted $\lfloor\frac{n-1}{3}\rfloor$ parties, it costs $O(\ell n + \lambda n^2)$ bits, which is asymptotically better than all previous results [9, 34] and optimal for sufficiently large input. Also it attains optimal running time and message complexity.

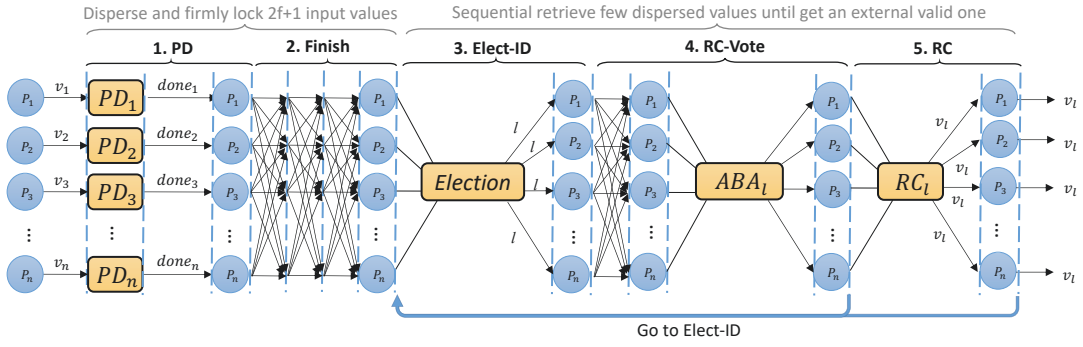


Figure 3.1: The execution flow of Dumbo-MVBA.

3.5.1 Overview of the Dumbo-MVBA protocol

As illustrated in Figure 4.8, the basic ideas of our Dumbo-MVBA protocol are: (i) the parties disperse their own input values through n concurrent PD instances, until they

consistently realize that enough *done*s proofs for the PD instances (i.e., $2n/3$) have been produced, so they can make sure that enough honest input values (i.e., $n/3$) have been firmly locked across the network; (ii) eventually, the parties can exchange *done*s proofs to explicitly stop all PD instances; (iii) then, the parties can invoke a common coin protocol Election to randomly elect a PD instance; (iv) later, the parties exchange their *lock* proofs of the elected PD instance and then leverage ABA to vote on whether to invoke the corresponding RC instance to recast the elected dispersal; (v) when ABA returns 1, all parties would activate the RC instance and might probably recast a common value that is externally valid; otherwise (i.e., either ABA returns 0 or RC recasts invalid value), they repeat Election, until an externally valid value is elected and collectively reconstructed.

3.5.2 Details of the Dumbo-MVBA protocol

Our Dumbo-MVBA protocol invokes the following modules: (i) asynchronous provable dispersal broadcast APDB := (PD, RC); (ii) asynchronous binary agreement ABA against adaptive adversary; (iii) $(f + 1, n)$ threshold signature with adaptive security; and (iv) adaptively secure $(2f + 1, n)$-Coin scheme (in the alias Election) that returns random numbers over $[n]$.

Each instance of the underlying modules can be tagged by a unique extended identifier ID. These explicit IDs extend id and are used to distinguish multiple activated instances of every underlying module. For instance, (PD[ID], RC[ID] represents a pair of (PD, RC) instance with identifier ID, where ID := $\langle \mathsf{id}, i \rangle$ extends the identification id to represent a specific APDB instance with a designated sender $\mathcal{P}_i$. Similarly, ABA[ID] represents an ABA instance with identifier ID, where ID := $\langle \mathsf{id}, k \rangle$ and $k \in \{1, 2, \dots\}$.

Protocol execution. Hereunder we are ready to present the detailed protocol description (as illustrated in Algorithm 4). Specifically, an Dumbo-MVBA instance with identifier id proceeds as: *Dispersal phase* (line 1-2, 13-18). The n parties activate n concurrent instances of the provable dispersal PD subprotocol. Each party $\mathcal{P}_i$ is the designated sender

Algorithm 4 Dumbo-MVBA protocol with identifier id and external Predicate, for each party $\mathcal{P}_i$: main process, cf. Alg. 5 for message handlers (a process that handles incoming messages and changes variables in Alg. 4 to take responses)

let *provens* $\leftarrow 0$, $RDY \leftarrow \{\,\}$
for each $j \in [n]$ **do**
  let $store[j] \leftarrow \emptyset$, $lock[j] \leftarrow \emptyset$, $rc\text{-}ballot[j] \leftarrow 0$
  initialize a provable dispersal instance PD$[\langle \mathsf{id}, j \rangle]$

1: **upon** receiving input v_i s.t. Predicate$(v_i) = true$ **do** ▷ dispersal phase
2: pass v_i into PD$[\langle \mathsf{id}, i \rangle]$ as input ▷ finish phase
3: **wait** for receiving any valid FINISH message
4: **for** each $k \in \{1, 2, 3, \dots\}$ **do**
5: $l \leftarrow$ Election$[\langle \mathsf{id}, k \rangle]$ ▷ elect-id phase
6: **if** $lock[l] \neq \emptyset$ **then**
7: **multicast** (RCBALLOTPREPARE, id, l, $lock$) ▷ recast-vote phase
8: **else multicast** (RCBALLOTPREPARE, id, l, $\perp$)
9: **wait** for receiving $2f + 1$ (RCBALLOTPREPARE, id, l, $\cdot$) messages from distinct parties or $rc\text{-}ballot[l] = 1$
10: $b \leftarrow$ ABA$[\langle \mathsf{id}, l \rangle](rc\text{-}ballot[l])$
11: **if** $b = 1$ **then** ▷ recast phase
12: $v_l \leftarrow$ RC$[\langle \mathsf{id}, l \rangle](store[l], lock[l])$
13: **if** Predicate$(v_l) = true$ **then** output v_l

of a particular PD instance PD$[\langle \mathsf{id}, i \rangle]$, through which $\mathcal{P}_i$ can disperse the coded fragments of its input v_i across the network.

Finish phase (line 3, 19-35). This has a three-round structure to allow all parties consistently quit PD instances. It begins when a sender produces the *done* proof for its PD instance and multicasts *done* to all parties through a DONE message, and finishes when all parties receive a FINISH message attesting that at least $2f + 1$ PD instances has been "done". In addition, once receiving valid FINISH, a party invokes *abandon*() to explicitly quit from all PD instances.

Elect-ID phase (line 5). Then all parties invoke the coin scheme Election, such that they obtain a common pseudo-random number l over $[n]$. The common coin l represents the identifier of a pair of (PD$[\langle \mathsf{id}, l \rangle]$, RC$[\langle \mathsf{id}, l \rangle]$) instances.

Recast-vote phase (line 6-9, 36-39). Upon obtaining the coin l, the parties attempt to agree on whether to invoke the RC$[\langle \mathsf{id}, l \rangle]$ instance or not. This phase has to cope with a major limit of RC subprotocol, that the RC$[\langle \mathsf{id}, l \rangle]$ instance requires all parties

Algorithm 5 Dumbo-MVBA protocol with identifier id and external Predicate, for each party $\mathcal{P}_i$: the protocol message handlers, cf. Alg. 4 for the main protocol process that might take responses if certain messages are received by this algorithm

upon PD[$\langle$id, $j\rangle$] delivers *store* **do** ▷ record *store*[j] for each PD[$\langle$id, $j\rangle$]
  store[j] $\leftarrow$ *store*

upon PD[$\langle$id, $j\rangle$] delivers *lock* **do** ▷ record *lock*[j] for each PD[$\langle$id, $j\rangle$]
  lock[j] $\leftarrow$ *lock*

upon PD[$\langle$id, $i\rangle$] delivers *done* **do** ▷ multicast proof *done* for PD[$\langle$id, $i\rangle$]
  multicast (Done, id, *done*)

upon receiving (Done, id, *done*) from party $\mathcal{P}_j$ for the first time **do**
  if ValidateDone($\langle$id, $j\rangle$, *done*) =1 **then**
    provens $\leftarrow$ *provens* + 1
    if *provens* = $n - f$ **then** ▷ $n - f$ Done $\Rightarrow$ $f + 1$ honest Done
      $\rho_{rdy,i} \leftarrow \mathsf{SignShare}_{(f+1)}(sk_i, \langle \textsc{Ready}, \mathsf{id}\rangle)$
      multicast (Ready, id, $\rho_{rdy,i}$) ▷ one honest Ready $\Rightarrow$ $n - f$ Done

upon receiving (Ready, id, $\rho_{rdy,j}$) from party $\mathcal{P}_j$ for the first time **do**
  if $\mathsf{VerifyShare}_{(f+1)}(\langle \textsc{Ready}, \mathsf{id}\rangle, (j, \rho_{rdy,j})) = 1$ **then**
    $RDY \leftarrow RDY \cup (j, \rho_{rdy,j})$
    if $|RDY| = f + 1$ **then** ▷ $f + 1$ Ready $\Rightarrow$ one honest Ready
      $\sigma_{rdy} \leftarrow \mathsf{Combine}_{(f+1)}(\langle \textsc{Ready}, \mathsf{id}\rangle, RDY)$
      multicast (Finish, id, σ_{rdy}) to all, if was not sent before

upon receiving (Finish, id, σ_{rdy}) from party $\mathcal{P}_j$ for the first time **do**
  if $\mathsf{VerifyThld}_{(f+1)}(\langle \textsc{Ready}, \mathsf{id}\rangle, \sigma_{rdy}) = 1$ **then** ▷ valid Finish $\Rightarrow$ $f + 1$ Ready
    abandon($\langle$id, $j\rangle$) for each $j \in [n]$
    multicast (Finish, id, σ_{rdy}) to all, if was not sent before
  else discard this invalid message

upon receiving (RcBallotPrepare, id, l, *lock*) from $\mathcal{P}_j$ **do**
  if $\mathsf{ValidateLock}(\langle \mathsf{id}, l\rangle, lock) = 1$ **then**
    lock[l] $\leftarrow$ *lock*
      ▷ *rc-ballot*[l] = 1 $\Rightarrow$ *lock*[l] is valid $\Rightarrow$ PD[$\langle$id, $j\rangle$] is recoverable
    rc-ballot[l] $\leftarrow$ 1

to invoke it to reconstruct a common value. To this end, the *recast-vote* phase is made of a two-step structure. First, each party multicasts its locally recorded $lock[l]$ through RcBallotPrepare message, if the $\mathsf{PD}[\langle \mathsf{id}, l \rangle]$ instance actually delivers $lock[l]$; otherwise, it multicasts $\perp$ through RcBallotPrepare message. Then, each party waits for up to $2f + 1$ RcBallotPrepare from distinct parties, if it sees valid $lock[l]$ in these messages, it immediately activates $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ with input 1, otherwise, it invokes $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ with input 0. The above design follows the idea of biased validated binary agreement presented by Cachin et al. in [34], and $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ must return 1 to each party, when $f + 1$ honest parties enter the phase with valid $lock[l]$.

Recast phase (line 10-12). When $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ returns 1, all honest parties would enter this phase and there is always at least one honest party has delivered the valid *lock* regarding $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$. As such, the parties can always invoke the corresponding $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ instance to reconstruct a common value v_l. In case the recast value v_l does not satisfy the external predicate, the parties can consistently go back to *elect-ID phase*, which is trivial because all parties have the same external predicate; otherwise, they output v_l.

3.5.3 Analyses of the Dumbo-MVBA protocol

Here we present the detailed proofs along with the complexity analyses for our Dumbo-MVBA construction.

Security intuition. The Dumbo-MVBA protocol described by Algorithm 4 solves asynchronous validate byzantine agreement among n parties against adaptive adversary controlling $f \leq \lfloor \frac{n-1}{3} \rfloor$ parties, given (i) adaptively secure f-resilient APDB protocol, (ii) adaptively secure f-resilient ABA protocol, (iii) adaptively secure $(f + 1, n)$-Coin protocol (in the random oracle model), and (iv) adaptively secure $(f + 1, n)$ threshold signatures. We highlight here the key intuitions as follows:

- Termination and safety of *finish phase*. If any honest party leaves the finish phase and enters the elect-ID phase, then: (i) all honest parties will leave the finish phase, and (ii) at least $2f + 1$ parties have produced *done* proofs for their dispersals.

- Termination and safety of *elect-ID phase*. Since the threshold of Election is $2f + 1$, $\mathcal{A}$ cannot learn which dispersals are elected to recover before $f + 1$ honest parties explicitly abandon all dispersals, which prevents the adaptive adversary from "tampering" the values dispersed by uncorrupted parties. Moreover, Election terminates in constant time.
- Termination and safety of *recast-vote* and *recast*. The honest parties would consistently obtain either 0 or 1 from recast-vote. If recast-vote returns 1, all parties invoke a RC instance to recast the elected dispersal, which will recast a common value to all parties. Those cost expected constant time.
- Quality of *recast-vote* and *recast*. The probability that *recast-vote* returns 1 is at least $2/3$. Moreover, conditioned on *recast-vote* returns 1, the probability that the *recast* phase returns an externally valid value is at least $1/2$.

The proofs of Dumbo-MVBA. Now we prove our Algorithm 4 satisfies all properties of MVBA with all but negligible probability.

Lemma 6. *Suppose a party $\mathcal{P}_l$ multicasts* (Done, id, *done*)*, where* ValidateDone($\langle$id, $l\rangle$, *done*) *=1. If all honest parties participate in the* ABA[$\langle$id, $l\rangle$] *instance, then the* ABA[$\langle$id, $l\rangle$] *returns 1 to all.*

Proof. If a party $\mathcal{P}_l$ did multicast a valid (Done, id, *done*), we know at least $f + 1$ honest parties delivers valid $lock[l]$ s.t. ValidateLock(ID, $lock[l]$) $= 1$, due to the *Provability* properties of APDB. Then according to the pseudocode of Algorithm 4, at least $f + 1$ honest parties will multicast valid (RcBallotPrepare, id, l, *lock*) to all. In this case, since all honest parties need to wait for $2f + 1$ RcBallotPrepare messages from distinct parties, then all honest parties must see a valid (RcBallotPrepare, id, l, *lock*) message. Therefore, all honest parties would input 1 to ABA[$\langle$id, $l\rangle$] instance. From the *validity* properties of the ABA protocol, we know that the ABA[$\langle$id, $l\rangle$] returns 1 to all. □

Lemma 7. *Suppose all honest parties participate the* $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ *instance and* $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ *return 1 to all. If all honest parties invoke* $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$*, then the* $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ *will return a same value to all honest parties. Besides, if* $\mathcal{P}_l$ *(sender) is an honest party, then the* $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ *will return an externally validated value.*

Proof. Since the $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ returns 1, we know at least one honest party inputs 1 to ABA, due to the *validity* properties of ABA. It also means that at least one honest party $\mathcal{P}_i$ receives a message $(\textsc{RcBallotPrepare}, \mathsf{id}, l, lock)$ which satisfies $\mathsf{ValidateLock}(\langle \mathsf{id}, l\rangle, lock) = 1$. According to the *Recast-ability* properties of APDB, all honest parties will terminate in $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$, and recover a common value, conditioned on all honest parties invoke $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$.

In addition, if $\mathcal{P}_l$ is an honest party, $\mathcal{P}_l$ always inputs an externally valid value to $\mathsf{PD}[\langle \mathsf{id}, l\rangle]$, due to the *Recast-ability* properties of APDB, the $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ will return the exactly same valid value to all parties. □

Lemma 8. *If an honest party invokes* $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$*, then at least* $2f + 1$ *distinct* PD *instances have completed, and all honest parties also invoked* $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$.

Proof. Suppose an honest party $\mathcal{P}_i$ invokes $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$, then it means that $\mathcal{P}_i$ receives a valid $\textsc{Finish}$ message. It also means that at least $f + 1$ parties multicast valid $\textsc{Ready}$ message, it implies that at least one honest party received $2f + 1$ valid $\textsc{Done}$ messages from distinct parties. Since each $\textsc{Done}$ message can verify the PD instance is indeed completed, at least $2f + 1$ distinct PD instances have been completed.

In addition, for $k = 1$, before an honest party invokes $\mathsf{Election}[\langle \mathsf{id}, 1\rangle]$, it must multicast the valid $\textsc{Finish}$ message, if it was not sent before. For the other honest parties, they will also invoke the $\mathsf{Election}[\langle \mathsf{id}, 1\rangle]$, upon receiving a valid $\textsc{Finish}$ message. For $k > 1$, without loss of generality, suppose an honest party $\mathcal{P}_i$ halts after invoking $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$, and another honest party $\mathcal{P}_j$ halts after invoking $\mathsf{Election}[\langle \mathsf{id}, k'\rangle]$, where $k' > k$. However, according the *agreement* of ABA, all honest parties will output the same bit 0 (not recast) or 1 (to recast); in addition, according to the *recast-ability* properties of APDB, all honest

parties will recover the same value if ABA returns 1. So, if $\mathcal{P}_i$ halts after invoking Election[$\langle$id, $k\rangle$], $\mathcal{P}_j$ shall also halt after invoking Election[$\langle$id, $k\rangle$]. Hence, the honest party $\mathcal{P}_j$ would not enter Election[$\langle$id, $k'\rangle$], when another honest party $\mathcal{P}_i$ would not invoke Election[$\langle$id, $k'\rangle$]. □

Lemma 9. Termination. *If every honest party $\mathcal{P}_i$ activates the protocol on identification* id *with proposing an input value v_i such that* Predicate(v_i) = *true, then every honest party outputs a value v for* id *in constant time.*

Proof. According to Algorithm 4, the Dumbo-MVBA protocol first executes n concurrent PD instances. Since all honest parties start with externally valid values and all massages sent among honest parties have been delivered, from the *termination* of APDB, if no honest party abandons the PD, any honest parties can know at least $n - f$ PD instances have completed; if any honest party abandons the PD instances, it means that this party has seen a valid Finish messages, which attests at least $n - f$ PD instances have completed.

When an honest party learns at least $n - f$ PD instances have completed, it will invoke Election[$\langle$id, $k\rangle$] to elect a random number l. From Lemma 8, we know all other honest parties also will invoke Election[$\langle$id, $k\rangle$]. In addition, all honest parties will input a value to ABA[$\langle$id, $l\rangle$], from the *termination* and *agreement* properties of ABA, the ABA[$\langle$id, $l\rangle$] will return a same value to all. Next, let us consider three following cases:

- **Case 1:** ABA[$\langle$id, $l\rangle$] returns 1 to all. According to the *recast-ability* properties of APDB, the RC[$\langle$id, $l\rangle$] instance will terminate and recover a same value to all. The recast value can be valid and satisfy the global Predicate, then this value will be decided as output by all parties.
- **Case 2:** ABA[$\langle$id, $l\rangle$] returns 1 to all. Due to the *recast-ability* of APDB, the RC[$\langle$id, $l\rangle$] instance will terminate and recover a same value to all. The value can be invalid due to the global external Predicate, the honest parties will repeat Election, until Case 1 occasionally happens.

- **Case 3:** If $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ returns 0 to all, then the honest parties will repeat Election, until Case 1 occasionally happens.

Now, we prove that the protocol terminates, after sequentially repeating ABA (and RC). Recall all honest parties start with dispersing externally valid values, so after $\mathsf{Election}[\langle \mathsf{id}, k \rangle]$ returns l for every $k \geq 1$, the probability that $\mathcal{P}_l$ is honest and completes $\mathsf{PD}[\langle \mathsf{id}, l \rangle]$ is at least $p = 1/3$. Due to the *unbiasedness* of Election, the coin ℓ returned by Election is uniform over $[n]$.

As such, let the event E_k represent that the protocol does not terminate when $\mathsf{Election}[\langle \mathsf{id}, k \rangle]$ has been invoked, so the probability of the event E_k, $\Pr[E_k] \leq (1-p)^k$. It is clear to see $\Pr[E_k] \leq (1-p)^k \to 0$ when $k \to \infty$, so the protocol eventually halts. Moreover, let K to be the random variable that the protocol just terminates when $k = K$, so $\mathbb{E}[K] \leq \sum_{K=1}^{\infty} K(1-p)^{K-1}p = 1/p = 3$, indicating the protocol terminates in expected constant time. □

Lemma 10. External-Validity. *If an honest party outputs a value v for* id*, then* $\mathsf{Predicate}(v) = true$.

Proof. According to Algorithm 4, when an honest party outputs a value v, there is always $\mathsf{Predicate}(v) = true$. Therefore, the external-validity trivially holds. □

Lemma 11. Agreement. *If any two honest parties output v and v' for* id *respectively, then* $v = v'$.

Proof. From lemma 8, we know if an honest party invokes $\mathsf{Election}[\langle \mathsf{id}, k \rangle]$, then all honest parties also invoke $\mathsf{Election}[\langle \mathsf{id}, k \rangle]$. From the *agreement* properties of Election, all honest parties get the same coin l. Hence, all honest parties will participate in the same $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ instance. Besides, due to the *agreement* of ABA, all honest parties will get a same bit b. Hence, upon $\mathsf{ABA}[\langle \mathsf{id}, l \rangle] = 1$, then all honest parties will participate in the same $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ instance. According to the *recast-ability* property of APDB, all honest parties must output the same value. □

Lemma 12. Quality. *If an honest party outputs v for* id*, the probability that v was proposed by the adversary is at most 1/2.*

Proof. Due to Lemma 8, as long as an honest party activates Election, at least $2f+1$ distinct PD instances have completed, which means these PD instances' senders can produce valid completeness *done* proofs. Moreover, if any honest party invokes Election$[\langle \mathsf{id}, k\rangle]$, all honest parties will eventually invoke Election$[\langle \mathsf{id}, k\rangle]$ as well. Suppose Election$[\langle \mathsf{id}, k\rangle]$ returns l, then all honest parties will participate in the ABA$[\langle \mathsf{id}, l\rangle]$ instance. If the sender $\mathcal{P}_l$ has completed the PD protocol, due to Lemma 6, the ABA$[\langle \mathsf{id}, l\rangle]$ will return 1 to all.

Then, if ABA$[\langle \mathsf{id}, l\rangle]$ returns 0, all parties will go to the next iteration to enter Election$[\langle \mathsf{id}, k+1\rangle]$; otherwise, ABA$[\langle \mathsf{id}, l\rangle]$ returns 1, all honest parties will participate in the RC$[\langle \mathsf{id}, l\rangle]$ instance, and the RC$[\langle \mathsf{id}, l\rangle]$ instance will return a common value to all parties, due to Lemma 7.

Let $\mathbb{P}_a$ to denote the set of the parties that are already corrupted by the adversary, when the adversary can tell the output of Election with non-negligible probability. Due to the *unpredictability* property of Election, upon the adversary can realize the output of Election, at least $f+1$ honest parties have already activated Election and therefore have abandoned all PD instances. This further implies that, once the adversary realizes the output of Election, the adversary can no longer disperse adversarial values by adaptively corrupting any so-far-uncorrupted senders outside $\mathbb{P}_a$.

Moreover, when the adversary is able to predicate the output of Election, at least $2f+1$ PD instances have been completed, out of which at most $|\mathbb{P}_a|$ instances are dispersed by the adversary. Therefore, we consider the worst case that: (i) only $f+1$ honest parties have completed their PD instances, and (ii) $|\mathbb{P}_a| = f$ and these f PD instances sent by the adversary have completed. In addition, due to the *unbiasedness* property of Election, the adversary cannot bias the distribution of the output of Election. So Election$[\langle \mathsf{id}, k\rangle]$ returns a coin l that is uniformly sampled over $[n]$, which yields the next three cases for any $k \in \{1, 2, \dots\}$:

- **Case 1:** If the sender $\mathcal{P}_l$ has not completed the PD instance yet, and the ABA[$\langle$id, $l\rangle$] returns 0, then repeats Election, the probability of this case at most is $1/3$; in such the case, the protocol would go to Election to repeat;
- **Case 2:** If the sender $\mathcal{P}_l$ has completed the PD protocol and the sender' input was determined by the adversary (which might or might not be valid regarding the global predicate), the probability of this case at most is $1/3$;
- **Case 3:** If the sender $\mathcal{P}_l$ has completed the PD protocol and the sender' input was not determined by the adversary, the probability of this case at least is $1/3$;

Hence, the probability of deciding an output value v proposed by the adversary is at most $\sum_{k=1}^{\infty}(1/3)^k = 1/2$. □

Theorem 2. *In random oracle model, the protocol described by Algorithm 4 (*Dumbo-MVBA*) realizes asynchronous validate byzantine agreement among n parties against adaptive adversary controlling $f < n/3$ parties, given (i) f-resilient* APDB *protocol against adaptive adversary, (ii) f-resilient* ABA *protocol against adaptive adversary, and (iii) adaptively secure non-interactive $(2f + 1, n)$ and $(f + 1, n)$ threshold signatures.*

Proof. Lemma 9, 10, 11 and 12 complete the proof. □

The complexity analysis of Dumbo-MVBA. The Dumbo-MVBA achieves: (i) asymptotically optimal round and message complexities, and (ii) asymptotically optimal communicated bits $O(\ell n + \lambda n^2)$ for any input $\ell \geq \lambda n$.

According to the pseudocode of algorithm 4, the breakdown of its cost can be briefly summarized in the next five phases: (i) the dispersal phase that consists of the n concurrent PD instances; (ii) the finish phase which is made of three all-to-all multicasts of Done, Ready and Finish messages; (iii) the elect-ID phase where is an invocation of Election; (iv) the recast-vote phase that has one all-to-all multicast of RcBallotPrepare messages and an invocation of ABA instance; (v) the recast phase where is to executes an RC instance.

Due to the complexity analysis of APDB in section 3.4.3, we know the PD's message complexity is $O(n)$ and its communication complexity is $O(\ell + n\lambda)$; the RC's message complexity is $O(n^2)$ and its communication complexity is $O(n\ell + n^2\lambda)$. So the complexities of Dumbo-MVBA protocol can be summarized as:

- **Running time:** The protocol terminates in expected constant running time due to Lemma 9.
- **Message complexity:** In the dispersal phase, there are n PD instances, each of which incurs $O(n)$ messages. In the finish phase, there are three all-to-all multicasts, which costs $O(n^2)$ messages. In the elect phase, there is one common coin, that incurs $O(n^2)$ messages. In the rc-vote phase, there is one all-to-all multicast and one ABA instance, which incurs $O(n^2)$ messages. In recast phase, there is only one RC instance, thus yielding $O(n^2)$ messages. Moreover, the elect phase, the rc-vote-phase, and the recast phase would be repeated for expected 3 times. To sum up, the overall message complexity of the Dumbo-MVBA protocol is $O(n^2)$.
- **Communication complexity:** In the dispersal phase, there are n PD instances, each of which incurs $O(\ell + n\lambda)$ bits. In the finish phase, there are three all-to-all multicasts, which corresponds to $O(n^2)$ λ-bit messages. In the elect-ID phase, there is one common coin, that incurs $O(n^2\lambda)$ bits. In the recast-vote phase, there is one all-to-all multicast and one ABA instance, thus incurring $O(n^2)$ messages, each of which contains at most λ bits. In the recast phase, there is only one RC instance, thus yielding $O(n^2)$ messages, each of which contains at most $O(\ell + n\lambda)$ bits. Moreover, the elect phase and the rc-vote phase would be repeated for expected 3 times, and the recast phase would be repeated for expected 2 times. Hence, the communication complexity of the Dumbo-MVBA protocol is $O(n\ell + n^2\lambda)$.

Note if considering $\ell \geq O(n\lambda)$, the Dumbo-MVBA protocol realizes optimal communication complexity $O(n\ell)$.

3.6 Dumbo-MVBA⋆: A Generic Optimal MVBA Framework

The *dispersal-then-recast* methodology can also be applied to bootstrap any existing MVBA to realize optimal communication for sufficiently large input. We call this extension protocol Dumbo-MVBA⋆. The key idea is to invoke the underlying MVBA with taking as input the small-size proofs of APDB. Though Dumbo-MVBA⋆ is a "reduction" from MVBA to MVBA itself, an advanced module instead of more basic building block such as binary agreement, this self-bootstrap technique can better utilize MVBA to achieve a simple modular design as explained in Figure 3.2, and we note it does not require the full power of APDB (and thus can potentially remove the rounds of communication generating the *done* proof).

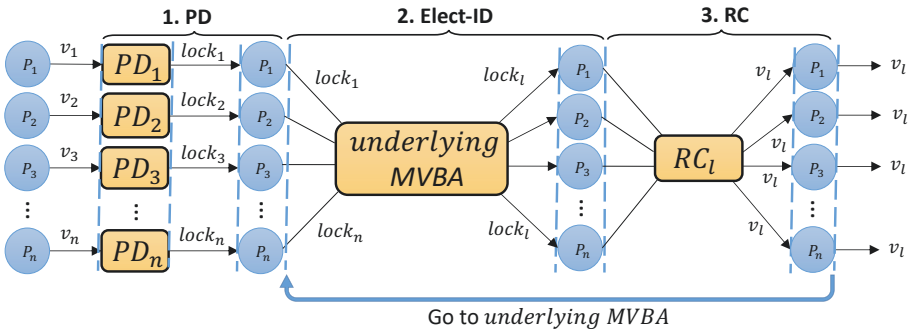


Figure 3.2: The execution flow of Dumbo-MVBA⋆.

3.6.1 Overview of the Dumbo-MVBA⋆ protocol

As shown in Figure 3.2, the generic framework still follows the idea of *dispersal-then-recast*: (i) each party disperses its own input value and obtains a *lock* proof attesting the recast-ability of its own dispersal; (ii) then, the parties can invoke any existing MVBA as a black-box to "elect" a valid *lock* proof, and then recover the already-dispersed value, until all parties recast and decide an externally valid value.

This generic Dumbo-MVBA⋆ framework presents a simple modular design that can enhance any existing MVBA protocol to achieve optimal communication for sufficiently large input, without scarifying the message complexity and running time of underlying MVBA. In particular, when instantiating the framework with using the MVBA protocol

due to Abraham et al. [9], we can obtain an optimal MVBA protocol that outperforms the state-of-the-art, since it achieves only $O(n\ell + n^2\lambda)$ communicated bits, without giving up the optimal running time and message complexity.

Algorithm 6 The Dumbo-MVBA⋆ protocol with identification id and external Predicate(), for each party $\mathcal{P}_i$

let $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k\rangle]$ to be an MVBA instance which takes as input string $lockproof$ and is parameterized by the next external predicate:

$$\mathsf{Predicate}_{\mathsf{Election}}(lockproof) \equiv (lockproof \text{ can be parsed as } \langle i, lock_i\rangle) \wedge (\mathsf{ValidateLock}(\langle \mathsf{id}, i\rangle, lock_i) \ \wedge \ i \in [n])$$

for each $j \in [n]$ **do**
  let $store[j] \leftarrow \emptyset$ and **initialize** an instance $\mathsf{PD}[\langle \mathsf{id}, j\rangle]$

1: **upon** receiving input v_i s.t. $\mathsf{Predicate}(v_i) = 1$ **do**
▷ provable dispersal phase
2: pass v_i into $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$ as input
3: **wait** for $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$ delivers $lock_i$
4: **for** each $k \in \{1, 2, 3, \dots\}$ **do**
▷ elect a finished dispersal to recast
5: $\langle l, lock_l\rangle \leftarrow \mathsf{MVBA}_{under}[\langle \mathsf{id}, k\rangle](\langle i, lock_i\rangle)$
6: $v_l \leftarrow \mathsf{RC}[\langle \mathsf{id}, l\rangle](store[l], lock_l)$
7: **if** $\mathsf{Predicate}(v_l) = true$ **then** output v_l

8: **upon** $\mathsf{PD}[\langle \mathsf{id}, j\rangle]$ delivers $store$ **do** ▷ record $store[j]$ for each $\mathsf{PD}[\langle \mathsf{id}, j\rangle]$
9: $store[j] \leftarrow store$

3.6.2 Details of the Dumbo-MVBA⋆ protocol

Here is our generic Dumbo-MVBA⋆ framework. Informally, a Dumbo-MVBA⋆ instance with identification id (as illustrated in Algorithm 6) proceeds as:

1. *Dispersal phase* (line 1-3, 8-9). n concurrent PD instances are activated. Each party $\mathcal{P}_i$ is the designated sender of the instance $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$, through which $\mathcal{P}_i$ disperses its input's fragments across the network.

2. *Elect-ID phase* (line 4-5). As soon as the party $\mathcal{P}_i$ delivers $lock_i$ during its dispersal instance $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$, it takes the proof $lock_i$ as input to invoke a concrete MVBA instance with identifier $\langle \mathsf{id}, k\rangle$, where $k \in \{1, 2, \dots\}$. The external validity of

underlying MVBA instance is specified to output a valid $lock_l$ for any PD instance PD[$\langle$id, $l\rangle$].

3. *Recast phase* (line 6-7). Eventually, the MVBA[$\langle$id, $k\rangle$] instance returns to all parties a common $lock_l$ proof for the PD[$\langle$id, $l\rangle$] instance, namely, MVBA elects a party $\mathcal{P}_l$ to recover its dispersal. Then, all honest parties invoke RC[$\langle$id, $l\rangle$] to recover a common value v_l. If the recast v_l is not valid, every party $\mathcal{P}_i$ can realize locally due to the same global Predicate, so each $\mathcal{P}_i$ can consistently go back the *elect-ID phase* to repeat the election by running another MVBA[$\langle$id, $k+1\rangle$] instance with still passing $lock_i$ as input, until a valid v_l can be recovered by an elected RC[$\langle$id, $l\rangle$] instance.

3.6.3 Analyses of the Dumbo-MVBA⋆ protocol

Here we present the detailed proofs along with the complexity analyses for our Dumbo-MVBA⋆ construction.

Security intuition. The Dumbo-MVBA⋆ protocol described by Algorithm 6 realizes (optimal) MVBA among n parties against adaptive adversary controlling $f \leq \lfloor \frac{n-1}{3} \rfloor$ parties, given (i) f-resilient APDB protocol against adaptive adversary (with all properties but abandon-ability and provability), (ii) adaptively secure f-resilient MVBA protocol. The key intuitions of Dumbo-MVBA⋆ as follows:

- The repetition of the phase (2) and the phase (3) can terminate in expected constant time, as the quality of every underlying MVBA instance ensures that there is at least $1/2$ probability of electing a PD instance whose sender was not corrupted before invoking MVBA.
- As such, the probability of not recovering any externally valid value to halt exponentially decreases with the repetition of *elect-ID* and *recast*. Hence only few (i.e., two) underlying MVBA instances and RC instances will be executed on average.

The proofs of Dumbo-MVBA⋆. Now we prove that Algorithm 6 satisfies all properties of MVBA except with negligible probability.

Lemma 13. *Suppose a party $\mathcal{P}_i$ delivers $\langle l, lock_l \rangle$ in any* $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ *that* $k \in [n]$*, then all honest parties would invoke* $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ *and recover a common value from* $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$*. Besides, if $\mathcal{P}_l$ (i.e., the sender of* $\mathsf{PD}[\langle \mathsf{id}, l \rangle]$*) was not corrupted before $lock_l$ was delivered, then the* $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ *returns a validated value.*

Proof. If any honest party delivers $\langle l, lock_l \rangle$ in any MVBA_{under} instance, all honest parties deliver the same $\langle l, lock_l \rangle$ in this MVBA_{under} instance, so all honest parties would invoke $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$. Moreover, due to the specification of $\mathsf{Predicate}_{\mathsf{Election}}$ shown in Algorithm 6, all honest parties deliver $\langle l, lock_l \rangle$, s.t. $\mathsf{ValidateLock}(\langle \mathsf{id}, l \rangle, lock_l) = 1$. According to the *recast-ability* property of APDB, all honest parties (that invoke $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$) will terminate and output the same value (or the same $\perp$). In addition, the *recast-ability* property also states: conditioned on that $\mathcal{P}_l$ was not corrupted before delivering $lock_l$ and it took as input a valid value v_l to disperse in $\mathsf{PD}[\langle \mathsf{id}, l \rangle]$, the $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ will return to all parties the valid value v_l. □

Lemma 14. Termination. *If every honest party $\mathcal{P}_i$ activates the protocol on identification* id *with proposing an input value v_i such that* $\mathsf{Predicate}(v_i) = true$*, then every honest party outputs a value v for* id*. Moreover, if the expected running time of the underlying* MVBA_{under} *is* $O(\mathsf{poly}_{rt}(n))$*,* Dumbo-MVBA⋆ *is expected to run in* $O(\mathsf{poly}_{rt}(n))$ *time.*

Proof. According to Algorithm 6, Dumbo-MVBA⋆ firstly executes n concurrent PD instance. From the *termination* of APDB: if a sender $\mathcal{P}_s$ is honest and all honest parties activate $\mathsf{PD}[\langle \mathsf{id}, s \rangle]$ without abandoning, then the honest sender $\mathcal{P}_s$ can deliver $lock_s$ for identification $\langle \mathsf{id}, s \rangle$ s.t. $\mathsf{ValidateLock}(\langle \mathsf{id}, s \rangle, lock_s) = 1$.

In case every honest party $\mathcal{P}_i$ passes an input to its PD instance, all honest parties can deliver a lock proof *lock* from PD, which satisfies the $\mathsf{Predicate}_{\mathsf{Election}}$ of $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$. Hence, each honest party $\mathcal{P}_i$ will pass a valid $\langle i, lock_i \rangle$ as input into $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$

for each iteration $k \in [n]$. Following the *agreement* and *termination* of MVBA, all honest parties can get the same output $\langle l, lock_l \rangle$ from each $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ instance.

Due to the *external-validity* of the underlying MVBA, the output $\langle l, lock_l \rangle$ of each $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ shall satisfy $\mathsf{Predicate}_{\mathsf{Election}}(\mathsf{id}, l, lock_l) = 1$. After $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ returns $\langle l, lock_l \rangle$, the $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ will be invoked and return a same value v_l to all in constant time due to Lemma 13. Let us consider two cases for any $k \in \{1, 2, \dots\}$ as follows:

- **Case 1:** If the value v_l returned by $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ is valid, then output the value.
- **Case 2:** If the value v_l returned by $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ is not valid, the parties will go back to the elect-ID phase to execute $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k+1 \rangle]$, until a valid value will be decided.

Now, we prove that the honest parties would terminate in expected constant time, except with negligible probability. Due to the *quality* properties of the MVBA, the probability that $\langle l, lock_l \rangle$ was proposed by the adversary is at most $1/2$ for each MVBA_{under} instance with different identification $\langle \mathsf{id}, k \rangle$. In addition, due to the *recast-ability* of APDB, whenever $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$'s output $\langle l, lock_l \rangle$ was not proposed by the adversary, a valid value can be collectively recovered by all honest parties due to $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$. So the probability that an externally valid v_l is recover after invoking each $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ is at least $p = 1/2$. Let the event E_k represent that the protocol does not terminate when $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ has been invoked for k times, so the probability of the event E_k, $\Pr[E_k] \leq (1-p)^k$. It is clear to see $\Pr[E_k] \leq (1-p)^k \to 0$ when $k \to \infty$, so the protocol eventually halts. Moreover, let K to be the random variable that the protocol just terminates when $k = K$, so $\mathbb{E}[K] \leq \sum_{K=1}^{\infty} K(1-p)^{K-1}p = 1/p = 2$, indicating the protocol is expected to terminate after sequentially invoking $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ twice. □

Lemma 15. External-Validity. *If an honest party outputs a value v for* id*, then* $\mathsf{Predicate}(v) = true$.

Proof. According to Algorithm 6, when an honest party outputs a value, $\mathsf{Predicate}(v) = true$. Therefore, the external-validity trivially follows. □

Lemma 16. Agreement. *If any two honest parties output v and v' for* id *respectively, then* $v = v'$.

Proof. From the *agreement* property of MVBA, all honest parties get the same output $\langle l, lock_l \rangle$. Hence, all honest parties will participate in the common RC$[\langle \mathsf{id}, l \rangle]$ instance. Moreover, due to the *recast-ability* property of APDB, all honest parties will recover the same value from each invoked RC$[\langle \mathsf{id}, l \rangle]$. In addition, all honest parties have the same a-priori known predicate, and they output only when the recast value from RC$[\langle \mathsf{id}, l \rangle]$ satisfying this global predicate. Thus the decided output of any two honest parties must be the same. □

Lemma 17. Quality. *If an honest party outputs v for* id*, the probability that v was proposed by the adversary is at most 1/2.*

Proof. Due to the *external-validity* and *agreement* properties of the underlying MVBA$_{under}$, every honest party can get the same output $\langle l, lock_l \rangle$ from MVBA$_{under}[\langle \mathsf{id}, k \rangle]$ which satisfies the *external* Predicate$_{\mathsf{Election}}$, namely, $lock_i$ is the valid lock proof for the sender $\mathcal{P}_l$'s dispersal instance due to ValidateLock$(\langle \mathsf{id}, l \rangle, lock_l) = true$.

Then, all honest parties will participate in the same RC$[\langle \mathsf{id}, l \rangle]$ instance, according to Algorithm 6. From Lemma 13, we know the RC$[\langle \mathsf{id}, l \rangle]$ will terminate and output a common value v_l to all. Because of the *quality* properties of the MVBA, the probability that $\langle l, lock_l \rangle$ was proposed by the adversary is at most 1/2. So RC$[\langle \mathsf{id}, l \rangle]$ returns a value v_l that might correspond the next two cases:

- **Case 1:** The sender $\mathcal{P}_l$ was corrupted by $\mathcal{A}$ (before delivering $lock_l$);
- **Case 2:** The sender $\mathcal{P}_l$ was not corrupted by $\mathcal{A}$ (before delivering $lock_l$), and executing RC$[\langle \mathsf{id}, l \rangle]$ must output the valid value proposed by this sender (when it was not corrupted), due to the *recast-ability* of APDB;

Due to the *fairness* of underlying MVBA_{under}, the probability of Case 1 is at most 1/2, while the probability of Case 2 is at least 1/2, so the probability of deciding a value v_l was proposed by the adversary is at most 1/2. □

Theorem 3. *The protocol described by Algorithm 6 (*Dumbo-MVBA⋆*) realizes asynchronous validate Byzantine agreement among n parties against adaptive adversary controlling $f < n/3$ parties, given (i) f-resilient* APDB *protocol against adaptive adversary, and (ii) f-resilient* MVBA *protocol against adaptive adversary.*

Proof. Lemma 14, 15, 16, and 17 complete the proof. □

The complexity analysis of Dumbo-MVBA⋆. According to the pseudocode of Algorithm 6, the cost of Dumbo-MVBA⋆ is incurred in the next three phase: (i) the dispersal phase consisting of n concurrent PD instances; (ii) the elect-ID phase consisting of few expected constant number (i.e., two) of underlying MVBA instances; (iii) the recast phase consisting of few expected constant number (i.e., two) of RC instances.

Recall the complexities of PD and RC protocols: PD costs $O(n)$ messages, $O(\ell + n\lambda)$ bits, and $O(1)$ running time; RC costs $O(n^2)$ messages, $O(n\ell + n^2\lambda)$ bits, and $O(1)$ running time. Suppose the underlying MVBA module incurs expected $O(\mathsf{poly}_{rt}(n))$ running time, expected $O(\mathsf{poly}_{mc}(n))$ messages, and expected $O(\mathsf{poly}_{cc}(\ell, \lambda, n))$ bits, where $O(\mathsf{poly}_{mc}(n)) \geq O(n^2)$ and $O(\mathsf{poly}_{cc}(\ell, \lambda, n)) \geq O(\ell n + n^2)$ due to the lower bounds of adaptively secure MVBA. Thus, the complexities of Dumbo-MVBA⋆ can be summarized as:

- **Running time:** Since PD and RC are deterministic protocols with constant running timing, the running time of Dumbo-MVBA⋆ is dominated by the underlying MVBA module, namely, $O(\mathsf{poly}_{rt}(n))$.
- **Message complexity:** The message complexity of n PD instances (or a RC instance) is $O(n^2)$. The message complexity of the underlying MVBA is $O(\mathsf{poly}_{mc}(n))$, where $O(\mathsf{poly}_{mc}(n)) \geq O(n^2)$. As such, the messages complexity of Dumbo-MVBA⋆ is dominated by the underlying MVBA protocol, namely, $O(\mathsf{poly}_{mc}(n))$.

- **Communication complexity:** The communication of n concurrent PD instances (or a RC instance) is $O(n\ell+n^2\lambda)$. The underlying MVBA module incurs $O(\mathsf{poly}_{cc}(\lambda,\lambda,n))$ bits. So the overall communication complexity of Dumbo-MVBA$\star$ is $O(\ell n + \lambda n^2 + \mathsf{poly}_{cc}(\lambda,\lambda,n))$.

As shown in Table 3.1, Dumbo-MVBA$\star$ reduces the communication of the underlying MVBA from $O(\mathsf{poly}_{cc}(\ell,\lambda,n))$ to $O(\ell n+\lambda n^2+\mathsf{poly}_{cc}(\lambda,\lambda,n))$, which removes all superlinear terms factored by ℓ in the communication complexity. In particular, for sufficiently large input whose length $\ell \geq \max(\lambda n, \mathsf{poly}_{cc}(\lambda,\lambda,n)/n)$, Dumbo-MVBA$\star$ coincides with the asymptotically *optimal* $O(n\ell)$ communication.

Table 3.1: Asymptotic performance of MVBA protocols for ℓ-bit inputs

Protocols	Running time	Message Comp.	Comm. Comp. (bits)
underlying MVBA	$O(\mathsf{poly}_{rt}(n))$	$O(\mathsf{poly}_{mc}(n))$	$O(\mathsf{poly}_{cc}(\ell,\lambda,n))$
Dumbo-MVBA$\star$	$O(\mathsf{poly}_{rt}(n))$	$O(\mathsf{poly}_{mc}(n))$	$O(\ell n + \lambda n^2 + \mathsf{poly}_{cc}(\lambda,\lambda,n))$

Concrete instantiation. Dumbo-MVBA$\star$ can be instantiated by extending the MVBA protocol of Abraham et al. [9]. Moreover, the MVBA protocol of Abraham et al. achieved expected $O(1)$ running time, $O(n^2)$ messages and $O(n^2\ell + n^2\lambda)$ bits, it's clear that our Dumbo-MVBA$\star$ framework can extend their result to attain $O(n\ell + n^2\lambda)$ bits without scarifying the optimal running time and message complexity. Note if considering $O(\ell) \geq O(n\lambda)$, the above instantiation of Dumbo-MVBA$\star$ realizes optimal $O(n\ell)$ communication complexity.

3.7 Optimal Asynchronous Atomic Broadcast

Based on Dumbo-MVBA protocols, it is straightforward to construct efficient asynchronous atomic broadcast (ABC) through improving Cachin et al.'s results in [34]. This Section would elaborate how to obtain these improvements.

3.7.1 Optimal ACS through Dumbo-MVBA

As discussed in Introduction, ACS is usually the intermediate "layer" to realize practical ABC [34, 92]. To construct ACS, some nice reductions MVBA were studied [34, 74] and have demonstrated real-world practicality. Hinted by those relevant studies, it becomes enticing to improve some MVBA-based ACS protocols (e.g., CKPS-ACS in [34]) by using Dumbo-MVBA to replace earlier burdensome MVBA building blocks, which might also cause improvements in ABC. Let us briefly review the syntax and properties of ACS.

Definition 4. *A protocol among n parties with maximal tolerance up to f adaptive corruption is said to be an asynchronous common subset (*ACS*) protocol, if it allows each parties to take as input a value and then collectively output a common subset of all the parties' input values. In addition, it satisfies the following properties, in the asynchronous authenticated message-passing model (c.f. Section 3.3), with all but except negligible probability:*

- **Agreement.** *If any two honest parties output, then their output sets must be same;*
- **Validity.** *If an honest party outputs a set S, then $|S| \geq n - f$ and S contains the input values from at least $n - 2f$ honest parties;*
- **Totality.** *If $n - f$ honest parties invoke the protocol with taking an input, then all honest parties can output.*

Recall the ACS construction due to Cachin et al. [34] (CKPS-ACS), which is a simple reduction from ACS to MVBA. Let Sign and Verify algorithms from EUF-CMA2 secure digital signature scheme. Assuming public key infrastructure, the public key pk_i of the corresponding party $\mathcal{P}_i$ is known by everyone in the system. CKPS-ACS takes the advantage of MVBA's external validity to output a set of $n - f$ message-signature pairs from distinct parties. As illustrated in Algorithm 7, the protocol can has two logical phases that proceed as follows:

- Message diffuse (line 1-6). Once a party receives an input value, it signs the value and broadcasts the value-signature pair to all parties; each party would wait for $2f + 1$ such value-signature pairs sent from distinct parties;
- Decide output (line 7-9). Each party proposes the set Q of value-signature pairs to an adaptively secure MVBA instance with a properly defined external predicate (e.g., denoted by MVBA_{acs}), and waits this MVBA instance to return a set Q' of $n - f$ value-signature pairs from distinct parties; then it can output S, namely, the values in Q'.

Algorithm 7 CKPS-ACS with identifier ID (for each party $\mathcal{P}_i$), excerpted from Fig 3 in [34]

let $Q = \emptyset$
let MVBA_{acs}[ID] to be an MVBA instance which takes as input Q and is parameterized by the next external predicate:
$\mathsf{Predicate}(Q) \equiv (Q$ can be parsed as $\{(j, v_j, \sigma_j)\}) \wedge (|Q| = n - f) \wedge (\forall\ (j, v_j, \sigma_j) \in Q,$
$\mathsf{Verify}(pk_j, \sigma_j, \langle \mathsf{ID}, v_j \rangle)) \wedge (\forall$ two $(j_1, v_{j_1}\sigma_{j_1})$ and $(j_2, v_{j_2}, \sigma_{j_2}) \in Q,\ j_1 \neq j_2).$

1: **upon** receiving input v_i **do**
2: $\quad \sigma_i \leftarrow \mathsf{Sign}(sk_i, \langle \mathsf{ID}, v_i \rangle)$
3: $\quad$ **multicast** (DIFFUSE, ID, v_i, σ_i) to all parties
4: **upon** receive (DIFFUSE, ID, v_j, σ_j) message from $\mathcal{P}_j$ for the first time **do**
5: $\quad$ **if** $\mathsf{Verify}(pk_j, \sigma_j, \langle \mathsf{ID}, v_j \rangle)$=1 **then**
6: $\quad\quad Q = Q \cup (j, v_j, \sigma_j)$
7: **upon** $|Q| = n - f$ **do**
8: $\quad Q' \leftarrow \mathsf{MVBA}_{acs}[\mathsf{ID}](Q)$ ▷ Here MVBA_{acs} is instantiated by Dumbo-MVBA protocol
9: $\quad$ output $S = \{v_j \mid (\cdot, v_j, \cdot) \in Q'\}$

The concrete performance of CKPS-ACS heavily depends on the actual instantiation of underlying MVBA_{acs}. Prior to this study, existing MVBA protocols [9, 34] have a ℓn^2-term in communication cost (ℓ is the input size of MVBA), thus resulting in burdensome cubic communicated bits during CKPS-ACS's execution. Nevertheless, thanks to the improvements achieved by our Dumbo-MVBA protocols, we can use Dumbo-MVBA directly to instantiate the underlying MVBA_{acs} to realize an Dumbo-MVBA improved ACS protocol (e.g., denoted by CKPS-ACS$^{\text{Dumbo-MVBA}}$ for short through the paper), thus improving the communication cost of this ACS construction by an $O(n)$ factor, so only expected quadratic bits would be sent (among honest parties). Besides, the Dumbo-MVBA improved CKPS-ACS protocol also remains optimal expected quadratic

message complexity, optimal expected constant running time, and the maximal tolerance against up to $n/3$ adaptive Byzantine corruption.

3.7.2 Analyses of the optimal ACS protocol

Here we present detailed complexity analyses for CKPS-ACS when using Dumbo-MVBA as the underlying MVBA_{acs} building block (denoted by CKPS-$\mathsf{ACS}^{\text{Dumbo-MVBA}}$ for short). In addition, since [34] did not abstract the functionality of Algorithm 7 as ACS and did not prove the algorithm satisfies Definition 4, we also give such proofs for sake of completeness.

Lemma 18. Agreement and totality. *Algorithm 7 satisfies the agreement and totality properties of* ACS *except with negligible probability.*

Proof. We prove agreement through proof by contradiction: due to lines 8 and 9, if Algorithm 7 does not satisfy agreement, the agreement of underlying MVBA_{acs} is also broken, which leads to a contradiction since MVBA_{acs} satisfies Definition 2.

There are at least $n - f$ parties that are honest through the course of the protocol. Conditioned on all honest parties start ACS, every honest party must receive a set of value-signature pairs satisfying MVBA_{acs}'s external validity condition. Hence, all honest parties would invoke MVBA_{acs} with passing externally valid input. Assuming Algorithm 7 might not satisfy totality, it would break the termination property of underlying MVBA_{acs}, leading to contradiction. □

Lemma 19. Validity. *If an honest party outputs a set S, then $|S| \geq n - f$ and S contains inputs from at least $n - 2f$ honest parties.*

Proof. Proof by contradiction: according to the external validity condition of MVBA_{acs} and the pseudocode of lines 8 and 9, if Algorithm 7 does not satisfy the validity property of ACS, then either the external validity of MVBA_{acs} or the unforgeability of digital signature is broken. □

Theorem 4. *In the authenticated setting, the protocol described by Algorithm 7 solves asynchronous common subset (*ACS*) among n parties against adaptive adversary controlling $f < n/3$ parties, given f-resilient* MVBA *protocol against adaptive adversary.*

Proof. Lemmas 18 and 19 complete the proof. □

The complexity analysis. The cost of Algorithm 7 is incurred in the next two phases: (i) everyone multicasts its digitally signed ACS input to all parties; (ii) all parties collectively execute a specific MVBA_{acs} instance with taking a set of $n - f$ message-signature pairs as input. We let MVBA_{acs} to be instantiated by our Dumbo-MVBA protocols. Considering the input length of ACS to be $|m|$, the complexities of CKPS-$\mathsf{ACS}^{\text{Dumbo-MVBA}}$ can be analyzed as follows:

- **Running time:** Since the multicasts are deterministic process with constant running timing, the running time of Algorithm 7 is dominated by underlying MVBA_{acs}. Recall we instantiate MVBA_{acs} by Dumbo-MVBA protocols, which enjoys asymptotically optimal constant running time. As such, CKPS-$\mathsf{ACS}^{\text{Dumbo-MVBA}}$ can terminate in expected constant time.
- **Message complexity:** The diffuse phase needs $O(n^2)$ messages, and MVBA_{acs} costs expected $O(n^2)$ if being instantiated by Dumbo-MVBA protocols. Hence, the messages complexity of CKPS-$\mathsf{ACS}^{\text{Dumbo-MVBA}}$ is $O(n^2)$.
- **Communication complexity:** The communication cost of the diffuse phase is $O(n^2\lambda + n^2|m|)$. The input length ℓ of MVBA_{acs} is $O(n\lambda + n|m|)$, so the underlying MVBA_{acs} incurs $O(n^2\lambda+n^2|m|)$ bits, since our optimal MVBA constructions incur only $O(n^2\lambda + n\ell)$ bits. The overall communication complexity of CKPS-$\mathsf{ACS}^{\text{Dumbo-MVBA}}$ is, therefore, $O(n^2\lambda + n^2|m|)$.

As shown in Table 3.2, CKPS-$\mathsf{ACS}^{\text{Dumbo-MVBA}}$ is asymptotically better than all other existing ACS protocols. We remark that all protocols listed in the table are adaptively secure against

maximal $n/3$ corruption (or can be trivially tuned against adaptive adversary by properly choosing adaptively secure cryptographic primitives).

Table 3.2: Asymptotic performance of ACS protocols among n parties with $|m|$-bit input and λ-bit security parameter.

Protocols	Running Time	Comm. Comp.	Message Comp.
ACS [92]	$O(\log n)$	$O(\lvert m\rvert n^2 + \lambda n^3 \log n)$	$O(n^3)$
ACS [74] †	$O(1)$	$O(\lvert m\rvert n^2 + \lambda n^3 \log n)$	$O(n^3)$
CKPS-ACS [34]	$O(1)$	$O(\lvert m\rvert n^3 + \lambda n^3)$	$O(n^2)$
CKPS-ACS$^{\text{AMS-MVBA}}$ ‡	$O(1)$	$O(\lvert m\rvert n^3 + \lambda n^3)$	$O(n^2)$
CKPS-ACS$^{\text{Dumbo-MVBA}}$	$O(1)$	$O(\lvert m\rvert n^2 + \lambda n^2)$	$O(n^2)$

† ACS in [74] also has an MVBA building block, but specifying this building block out of [34], [9] and this work would not make any asymptotic differences.

‡ CKPS-ACS [34] with AMS-MVBA [9] as the underlying MVBA_{acs} building block, called CKPS-ACS$^{\text{AMS-MVBA}}$.

3.7.3 Efficient and adaptively secure ABC

Now we are approaching to our end goal of efficient ABC with adaptive security as by-product[1]. Recall the syntax and needed properties of ABC as follows.

Definition 5. *In an asynchronous atomic broadcast protocol among n parties that can tolerate up to f adaptive corruptions, each party can keep on receiving a polynomial-bounded number of payload messages (from the adversary) as input and outputs some payload messages (which can be chronologically ordered as a sequence, c.f. [34]). Moreover, the following properties shall be satisfied, in the asynchronous authenticated setting (c.f. Section 3.3), except with negligible probability:*

- **Agreement.** *If an honest party outputs a message v, then all honest parties output v;*
- **Total order.** *If two parties output two sequences of payload messages $(v_0, v_1, \ldots, v_T)$ and $(v'_0, v'_1, \ldots, v'_{T'})$, respectively, then $v_t = v'_t$ for any $t \leq \min(T, T')$;*

[1]Note that in reality, it could be acceptable to choose more lightweight cryptographic primitives that are not adaptively secure to implement our approach for performance, see [58] for some relevant guidance.

- **Censorship resilience.** *If a payload message m is input to $n - f$ honest parties, then it is eventually output by all honest parties.*

ABC through Dumbo-MVBA improved ACS. There are two typical reductions from ABC to ACS. One was firstly described in [34], which sequentially executes ACS instances with taking payload messages as ACS input in a first-come-first-serve manner. The other one was recently invented in [92], which is more involved by using threshold encryption to allow each party to encrypt a random batch of payload messages and use the ciphertext as ACS input, thus improving communication efficiency without harming censorship resilience.

When using CKPS-ACS$^{\text{Dumbo-MVBA}}$ to replace ACS instantiations in [34] and [92], the obtained ABC protocols still satisfy Definition 7, and we omit such proofs as they can be found in [34] and [92]. Moreover, given adaptively secure ACS (and other adaptive secure cryptographic primitives), ABC constructions in [34] and [92] are both adaptively secure, so plugging CKPS-ACS$^{\text{Dumbo-MVBA}}$ also preserves the adaptive security.

The asymptotically improvement is clear: compared to using CKPS-ACS to build ABC, our result can save an order of $O(n)$ in communications; in contract to ACS in HoneyBadger BFT, this work reduces an $O(n)$ factor in messages and a $O(\log n)$ factor in running time; compared to ACS in Dumbo BFT, it saves an order of $O(n)$ in messages. All above asymptotic improvements can be obtained, disregarding the choice of the reduction from ABC to ACS.

3.8 Summary

In this chapter, we present the first MVBA protocols with expected $O(\ell n + \lambda n^2)$ communicated bits. Our results complement the recent breakthrough of Abraham et al. at PODC '19 [9] and solve the remaining part of the long-standing open problem from Cachin et al. at CRYPTO '01 [34]. More precisely, we showcase:

Theorem 5. *There exist protocols in the authenticated setting with setup assumptions and random oracle, such that it solves the* MVBA *problem [9, 34] among n parties against*

an adaptive adversary controlling up to $f \leq \lfloor \frac{n-1}{3} \rfloor$ *parties, with expected* $O(\ell n + \lambda n^2)$ *communicated bits and expected constant running time, where* ℓ *is the input length and* λ *is a cryptographic security parameter.*

Table 3.3: Asymptotic performance of MVBA protocols among n parties with ℓ-bit input and λ-bit security parameter.

Protocols	Communication (Bits)	Word†	Time	Message
Cachin et al. [34]‡	$O(\ell n^2 + \lambda n^2 + n^3)$	$O(n^3)$	$O(1)$	$O(n^2)$
Abraham et al. [9]	$O(\ell n^2 + \lambda n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$
Our Dumbo-MVBA	$O(\ell n + \lambda n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$
Our Dumbo-MVBA⋆	$O(\ell n + \lambda n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$

† [9] defines a word to contain a constant number of signatures/inputs. The word communication measures the expected total number of words sent by honest parties.
‡ [34] realizes that their construction can be generalized against adaptive adversary, when given threshold cryptosystems with adaptive security.

Our result not only improves communication complexity upon [9, 34] as illustrated in Table 3.3, but also is *optimal* in the asynchronous setting regarding the following performance metrics:

1. The execution incurs $O(\ell n + \lambda n^2)$ bits on average, which coincides with the *optimal communication* $O(\ell n)$ when $\ell \geq O(\lambda n)$. This optimality can be seen trivially, since each honest party has to receive the ℓ-bit output, indicating a minimum communication of $\Omega(\ell n)$ bits.

2. As [9], it can tolerate an *adaptive adversary* controlling up to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine parties, which achieves the *optimal resilience* in the asynchronous network according to the upper bound of resilience presented by Bracha [30].

3. Same to [9, 34], it terminates in expected constant asynchronous rounds with overwhelming probability, which is essentially *asymptotically optimal* for asynchronous BA [21, 60].

4. As [9, 34], it attains *asymptotically optimal* $O(n^2)$ messages, which meets the lower bound of the messages of optimally-resilient asynchronous BA against adaptive adversary [4, 9].

Furthermore, our MVBA protocols can immediately be applied to construct efficient asynchronous atomic broadcast with reduced communication blow-up as previously suggested in [34]. Moreover, they can provide better building blocks for these asynchronous BFT protocols [73, 74, 87], the recent constructions of practical asynchronous atomic broadcast that use MVBA at their core in order to achieve high levels of efficiency.

CHAPTER 4

DUMBO-NG: FAST ASYNCHRONOUS BFT CONSENSUS WITH THROUGHPUT-OBLIVIOUS LATENCY

This chapter presents Dumbo-NG, which can approach the maximum throughput without trading latency, i.e., realize throughput-oblivious latency. As a result, it achieves high throughput, low latency and guaranteed censorship resilience simultaneously.

4.1 Background

The huge success of Bitcoin [100] and blockchain [32, 39] leads to an increasing tendency to lay down the infrastructure of distributed ledger for mission-critical applications. Such decentralized business is envisioned as critical global infrastructure maintained by a set of mutually distrustful and geologically distributed nodes [20], and thus calls for consensus protocols that are both secure and efficient for deployment over the Internet.

Asynchronous BFT for indispensable robustness. The consensus of decentralized infrastructure has to thrive in a highly adversarial environment. In particular, when the applications atop it are critical financial and banking services, some nodes can be well motivated to collude and launch malicious attacks. Even worse, the unstable Internet might become part of the attack surface due to network fluctuations, misconfigurations and even network attacks. To cope with the adversarial deployment environment, *asynchronous* Byzantine-fault tolerant (BFT) consensuses [9, 34, 58, 74, 88, 92] are arguably the most suitable candidates. They can realize high security-assurance to ensure liveness (as well as safety) despite an asynchronous adversary that can arbitrarily delay messages. In contrast, many (partial) synchronous consensus protocols [11, 12, 16, 24, 44, 71, 72, 105, 122] such as PBFT [41] and HotStuff [125] might sustain the inherent *loss of liveness* (i.e., generate unbounded communications without making any progress) [60, 92] when unluckily encountering an asynchronous network adversary.

4.1.1 Motivation

Unfortunately, it is fundamentally challenging to realize practical asynchronous BFT consensus, and none of such protocols was widely adopted due to serious efficiency concerns. The seminal FLP "impossibility" [60] proves that no *deterministic* consensus exists in the asynchronous network. Since the 1980s, many attempts [6, 21, 22, 35, 40, 107, 111] aimed at to circumventing the "impossibility" by *randomized* protocols, but most of them focused on theoretical feasibility, and unsurprisingly, several attempts of implementations [37, 97] had inferior performance.

Until recently, the work of HoneyBadger BFT (HBBFT) demonstrated the first asynchronous BFT consensuses that is performant in the wide-area network [92]. As shown in Figure 4.1, HBBFT was instantiated by adapting the classic asynchronous common subset (ACS) protocol of Ben-Or et al. [23]. It firstly starts n parallel reliable broadcasts (RBCs) with distinct senders. Here n is the total number of nodes, and RBC [30] emulates a broadcast channel via point-to-point links to allow a designated sender to disseminate a batch of input transactions. However, we cannot ensure that every honest node completes a certain RBC after a certain time due to asynchrony. So an agreement phase is invoked to select $n - f$ completed and common RBCs (where f is the number of allowed faulty nodes). In HBBFT, the agreement phase consists of n concurrent asynchronous binary Byzantine agreement (ABA). Each ABA corresponds to a RBC, and would output 1 (resp. 0) to solicit (resp. omit) the corresponding RBC in the final ACS output.

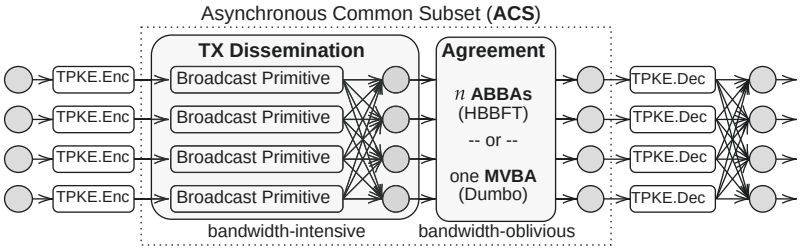


Figure 4.1: Execution flow of an epoch in HBBFT, Dumbo and their variants. The protocols proceed by consecutive epochs.

The above ACS design separates the protocol into *bandwidth-intensive* broadcast phase and *bandwidth-oblivious* agreement phase. Here the broadcast is *bandwidth-intensive* (i.e.,

latency heavily relies on available bandwidth), because of disseminating a large volume of transactions; and the agreement is *bandwidth-oblivious* (i.e., latency depends on network prorogation delay more than bandwidth), as it only exchanges a few rounds of short messages. HBBFT then focused on optimizing the bandwidth-intensive part—transaction broadcasts. It adapted the techniques of using erasure code and Merkle tree from verifiable information dispersal [38] to reduce the communication cost of Bracha's RBC [30], and realized amortized $O(n)$ communication complexity for sufficiently large input batch. As such, HBBFT can significantly increase throughput via batching more transactions, but its n concurrent ABAs incurred suboptimal expected $O(\log n)$ rounds. A recent work Dumbo [74] concentrated on the latency-critical part consisting of n ABAs, and used a single asynchronous multi-valued validated Byzantine agreement (MVBA) to replace the slow n ABAs. Here MVBA is another variant of asynchronous BA whose output satisfies a certain global predicate, and can be constructed from 2-3 ABAs (e.g., CKPS01 [34]) or from more compact structures (e.g., AMS19 [9] and GLL+22 [73]). Thanks to more efficient agreement phase based on MVBA, Dumbo reduced the execution rounds from expected $O(\log n)$ to $O(1)$, and achieved an order-of-magnitude of improvement on practical performance.

Actually, since HBBFT [92], a lot of renewed interests in addition to Dumbo are quickly gathered to seriously explore whether asynchronous protocols can ever be practical [9, 58, 69, 70, 77, 124]. Notwithstanding, few existing "performant" asynchronous BFT consensuses can realize high security assurance, low latency, and high throughput, simultaneously. Here down below we briefly reason two main practical obstacles in the cutting-edge designs.

Throughput that is severely hurting latency. A serious practicality hurdle of many existing asynchronous protocols (e.g. HBBFT and Dumbo) is that their maximum throughput is only achievable when their latency is sacrificed. As early as 2016, HBBFT [92] even explicitly argued that latency is dispensable for throughput and robustness, if aiming at the decentralized version of payment networks like VISA/SWIFT. The argument might be correct at the time of 2016, but after all these years, diverse decentralized

applications have been proposed from quick inter-continental transactions [46] to instant retail payments [20]. Hence it becomes unprecedentedly urgent to implement robust BFT consensus realizing high throughput while preserving low latency.

However, the existing performant asynchronous BFT protocols such as HBBFT and Dumbo consist of two main phases: the dissemination phase and the agreement phase. Each node broadcasts transactions in the dissemination phase that contribute to throughput; however, due to the agreement phase being an asynchronous protocol that needs multiple rounds to decide which transactions can be output, there is no contribution to throughput in this phase. Thus, in order to achieve maximum throughput, each node has to broadcast a huge batch of transactions during the agreement phase. Clearly, inferior latency is a natural consequence of larger batches. For example, Figure 4.2 clarifies: (i) when each node broadcasts a small batch of 1k tx in Dumbo (n=16), the latency is not that bad (2.39 sec), but the throughput is only 4,594 tx/sec, as the transaction dissemination only takes 16.5% of all running time; (ii) when the batch size increases to 30k tx, the broadcast of transactions possesses more than 50% of the running time, and the throughput becomes 37,620 tx/sec for better utilized bandwidth, but the latency dramatically grows to nearly 9 sec.

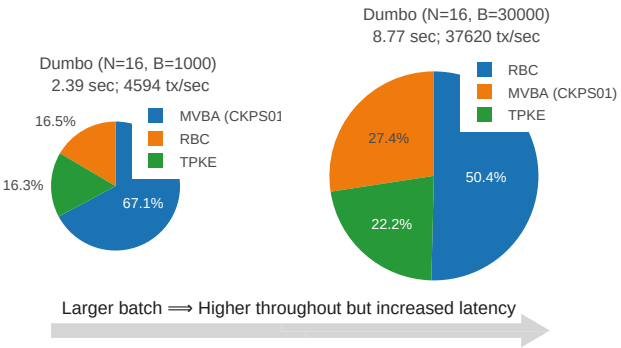


Figure 4.2: Latency breakdown of Dumbo (on 16 Amazon EC2 c5.large instances across different regions). $|B|$ is batch size, i.e., the number of tx to broadcast by each node (where each tx is 250-byte to approximate the size of Bitcoin's basic tx). TPKE is a technique from HBBFT for preventing censorship.

Liveness[1] **relies on heavy cryptography or degraded efficiency**. During the dissemination phase, an adversary might delay broadcasts containing transactions it dislikes. Hence, it may never output the f slowest broadcasts in an asynchronous environment. Faced with the potential of censorship, current solutions rely on asymptotically larger communications, expensive cryptographic operations, or possibly limitless memory. Specifically,

- One somewhat trivial "solution" to censorship-resilience is to diffuse transactions across all nodes and let every node work redundantly. Therefore, even if the adversary can slow down up to f honest nodes' broadcasts, it cannot censor a certain transaction tx, because other $n-2f$ honest nodes still process tx. This is the exact idea in Cachin et al.'s asynchronous atomic broadcast protocol [34], but clearly incurs another $O(n)$ factor in the communication complexity. Recently, Tusk [54] leveraged de-duplication technique to let a small number of k nodes process each transaction according to transaction hash [121] or due to the choice of clients. Here k is expected a small security parameter to luckily draw a fast and honest node. Nevertheless, an asynchronous adversary (even without actual faults) can prevent up to f honest nodes from eventually output in Tusk, and therefore the transactions duplicated to k nodes can still be censored unless $k \geq f+1$, i.e., $O(n)$ redundant communication still occurs in the worst case. That means, though de-duplication techniques are enticing, we still need underlying consensus stronger (e.g., any honest node's input must eventually output) to reduce redundant communication by these techniques without hurting liveness.

- As an alternative, HBBFT introduces threshold public key encryption (TPKE) to encrypt the broadcast input.[2] Now, transactions are confidential against the adversary

[1]Remark that liveness in the asynchronous setting cannot be guaranteed by merely ensuring protocols to progress without stuck. It needs to consider the liveness notion (e.g. validity from [34]) to ensure that any tx input by sufficient number of honest nodes must eventually output, which was widely adopted in [58, 70, 73, 74, 77, 92, 124].

[2]Remark that one also can use asynchronous verifiable secret sharing (AVSS) to replace TPKE, since both can implement a stronger consensus variant called casual broadcast [34], i.e., it first outputs transactions in a confidential manner, and then reveals. We do not realize any practical censorship-resilience implementation based on AVSS.

before they are solicited into the final output, so that the adversary cannot learn which broadcasts are necessary to delay for censoring a certain transaction. But TPKE decryption could be costly. Figure 4.2 shows that in some very small scales $n = 16$, TPKE decryption already takes about 20% of the overall latency in Dumbo (using the TPKE instantiation [17] same to HBBFT). For larger scales, the situation can be worse, because each node computes overall $O(n^2)$ operations for TPKE decryptions.

- Recently, DAG-Rider [77] presented a (potentially unimplementable) defense against censorship: the honest nodes do not kill the instances of the slowest f broadcasts but forever listen to their delivery. So if the slow broadcasts indeed have honest senders, the honest nodes can eventually receive them and then attempt to put them into the final consensus output. This intuitively can ensure all delayed broadcasts to finally output, but also incurs probably unbounded memory because of listening an unbounded number of broadcast instances that might never output due to corrupted senders (as pointed out by [54]).[3]

Given the state-of-the-art of existing "performant" asynchronous BFT consensuses, the following fundamental challenge remains:

> *Can we push asynchronous BFT consensus further to realize minimum latency, maximum throughput, and guaranteed censorship-resilience, simultaneously?*

4.1.2 Challenges

Here we take a brief tour to our solution, with explaining the main technical challenges and how we overcome the barriers.

In Dumbo-NG, we aim to make broadcast and agreement to execute completely concurrently. As Figure 4.3 illustrates, we let each node act as a sender in an ever-running

[3] In DAG-Rider [77], every node has to keep on listening unfinished broadcasts. So if there are some nodes get crashed or delayed for a long time, the honest nodes need to listen more and more unfinished broadcasts with the protocol execution. The trivial idea of killing unfinished broadcasts after some timeout would re-introduce censorship threat, because this might kill some unfinished broadcasts of slow but honest nodes.

multi-shot broadcast to disseminate its input transactions, and concurrently, run the agreement phase to pack the broadcasted transactions into the final consensus output. Now, the bandwidth-intensive transaction dissemination is continuously running to closely track the network capacity over all running time, and no longer needs to use large batch sizes to contend with the bandwidth-oblivious agreement modules for seizing network resources (as prior art does). As such, it becomes promising to obtain the peak throughput without hurting latency.

However, the seemingly simple idea of running broadcasts concurrent to Byzantine agreement (BA) is facing fundamental challenges in the asynchronous setting. Here we briefly overview the barriers and shed a light on our solution to tackle them.

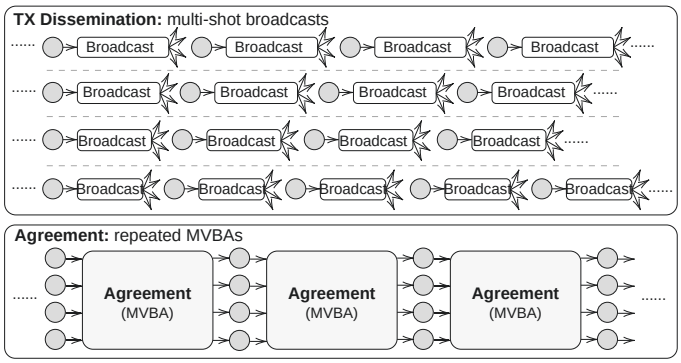


Figure 4.3: High-level of Dumbo-NG. Each node leads an ever-running multi-shot broadcast to disseminate its input transactions. Aside from broadcasts, a sequence of asynchronous multi-valued validated Byzantine agreements (MVBAs) are executed to totally order all broadcasted transactions.

Challenge I: allow BA *to pack broadcasts, concurrently & validly*. In an asynchronous network, the adversary can arbitrarily delay broadcasts, and thus some multi-shot broadcast might progress very fast while some might move forward much slower. So the concurrent Byzantine agreement modules have to agree on the "valid" progress of each multi-shot broadcast instance, where "valid" progress means the broadcast has indeed progressed up to here. The above task, intuitively, is much more challenging than the agreement problem in HBBFT/Dumbo (which only decides 1/0 for each single-shot broadcast to mark whether the broadcast is completed or not). At first glance, we seemingly need asynchronous BA with strong validity, because the agreed broadcast progress needs to be from some

honest node to ensure it was indeed completed (otherwise, the adversary can manipulate the agreement result to let honest nodes agree on some broadcast progresses that were not completed). But, unfortunately, strong validity is unimplementable for multi-valued agreement in the asynchronous setting, as it needs huge communication cost exponential in input length [61].

To circumvent the challenge, we carefully add quorum certificates to the multi-shot broadcasts by threshold signature, such that the adversary cannot forge a certificate for some uncompleted broadcast progress. In particular, our multi-shot broadcast can be thought of a compact variant of running a sequence of verifiable consistent broadcasts [113], in which a quorum certificate can prove that the honest nodes either have delivered (or can retrieve) the same sequence of all broadcasted transactions [34, 54]. This allows us to design the needed agreement module by (implementable) asynchronous MVBA with fine-tuned external validity. We let MVBA's input/output to be a vector of n broadcasts' certificates, and the external validity checks: (i) all n certificates are valid, (ii) at least $n - f$ certificates attest that their corresponding $n - f$ broadcasts have progressed. As such, we can run a sequence of MVBAs completely concurrent to the n ever-running broadcasts, and each MVBA can pack $n - f$ progressed broadcasts to form a final consensus output.

Challenge II: output all completed broadcasts to prevent censorship. Nevertheless, external validity of MVBA is not enough to ensure liveness, as it cannot guarantee that all progressed broadcasts can be solicited by some MVBA to output, and the censorship threat is still a valid concern. The reason behind the problem is: the conventional MVBA notion [34] allows the adversary to fully decide the agreed result (as long as satisfying the external validity condition), so in our context, the adversary can exclude up to f honest nodes' broadcasts from the final consensus output.

To overcome this subtle issue, we realize that some recent MVBA protocols [9, 73, 88] actually have an additional *quality* property (at no extra cost). Here *quality* means that with at least 1/2 probability (or other constant probability), the MVBA's output is proposed by some honest node. Hence, if we carefully choose an MVBA protocol with quality,

liveness (aka censorship-resilience) can be guaranteed because: once a broadcast's quorum certificate is received by all honest nodes, it will be decided to output after expected 2 MVBAs.

4.2 Related work

Recently, Tusk [54] adapted Prism's [18] core idea to separate transaction diffuse and agreement into the asynchronous setting, and presented how to diffuse transactions concurrently to a compact DAG-based asynchronous consensus: each node multicasts transaction batches to the whole network and waits $n - f$ naive receipt acknowledgements, such that the digests of transaction batches (instead of the actual transactions) can be agreed inside Tusk's DAG. Nevertheless, the above transaction diffuse does not generate quorum certificates for transaction retrievability by itself, but relies on consistent broadcasts inside Tusk's DAG to generate such certificates. That means, diffused transactions of f honest nodes might have no quorum certificates generated for retrievability in Tusk because their corresponding consistent broadcasts are never completed (and they will finally be garbage collected). In contrast, we require *every* node (even the slowest) can generate certificates for retrievability of its own input transactions through a multi-shot broadcast instance. This is critical for preventing censorship, because any honest node, no matter how slow it is, can generate these certificates and use them to convince the whole network to solicit its disseminated input into the final consensus output. This corresponds to the reason why Tusk's transaction diffuse cannot directly replace our transaction dissemination path without hurting censorship resilience.

Another recent work DispersedLedger [124] recognized that the agreement phase in HBBFT does not well utilize much bandwidth. It separates the bandwidth-intensive transaction dissemination phase into two parts: dispersal and retrieval. However, DispersedLedger still cannot achieve throughput-oblivious latency and effective censorship-resilience [63]. Besides closely related studies [54, 73, 74, 77, 92, 124] discussed in Introduction, there also exist a few works [80, 113] including some very recent ones [66, 87] that consider adding an optimistic "fastlane" to the slow asynchronous atomic broadcast.

The fastlane could simply be a fast leader-based deterministic protocol. This line of work is certainly interesting, however in the adversarial settings, the "fastlane" never succeeds, and the overall performance would be even worse than running the asynchronous atomic broadcast itself. This paper, on the contrary, aims to directly improve asynchronous BFT atomic broadcast, and can be used together with the optimistic technique to provide a better underlying pessimistic path. In addition, BEAT [58] cherry-picked constructions for each component in HBBFT (e.g., coin flipping and TPKE without pairing) to demonstrate better performance in various settings, and many of its findings can benefit us to choose concrete instantiations for the future production-level implementation. There are also interesting works on asynchronous distributed key generation [7, 55, 64, 79], which could be helpful to remove the private setup phase in all recent asynchronous BFT protocols.

In addition to fully asynchronous protocols, a seemingly feasible solution to robust BFT consensus is choosing a conservative upper bound of network delay in (partially) synchronous protocols. But this might bring serious performance degradation in latency, e.g., the exaggeratedly slow Bitcoin. Following the issue, a large number of "robust" (partially) synchronous protocols such as Prime [11], Spinning [122], RBFT [16] and many others [47, 48] are also subject to this robustness-latency trade-off. Let alone, none of them can have guaranteed liveness in a pure asynchronous network, inherently [60]. In addition, a few recent results [8, 105, 119] make synchronous protocols to attain fast (responsive) confirmation in certain good cases, but still suffer from slow confirmation in more general cases.

4.3 Problem Formulation

4.3.1 System model

We aim to design practical BFT consensus (atomic broadcast) protocols in the asynchronous setting. In short, we adopt a widely-adopted asynchronous message-passing model [9, 14, 34, 35, 58, 74, 88, 92] with setup assumptions. In greater detail, we consider:

Known identities & setup for threshold signature. There are n designated nodes in the system, each of which has a unique identity. W.o.l.g, their identities are denoted from $\mathcal{P}_1$ to $\mathcal{P}_n$. In addition, non-interactive threshold signature (TSIG) is properly set up, so all nodes can get and only get their own secret keys in addition to the public keys. The setup can be done through distributed key generation [7, 55, 56, 64, 68, 76, 79, 108] or a trusted dealer.

$n/3$ **Byzantine corruptions.** We consider that up to $f = \lfloor (n-1)/3 \rfloor$ nodes might be fully controlled by the adversary. Remark that our implementation might choose statically secure threshold signature as a building block for efficiency as same as other practical asynchronous protocols [58, 74, 92], noticing that a recent adaptively secure attempt [85] has dramatically degraded throughput less than half of its static counterpart for moderate scales ~50 nodes. Nevertheless, same to [9, 34, 88], our protocol can be adaptively secure to defend against an adversary that might corrupt nodes during the course of protocol execution, if given adaptively secure threshold signature [83,84] and MVBA [9,88]. Besides adaptively secure building blocks, the other cost of adaptive security is just an $O(n)$-factor communication blow-up in some extreme cases.

Asynchronous fully-meshed point-to-point network. We consider an asynchronous message-passing network made of fully meshed authenticated point-to-point (p2p) channels [14]. The adversary can arbitrarily delay and reorder messages, but any message sent between honest nodes will eventually be delivered to the destination without tampering, i.e., the adversary cannot drop or modify the messages sent between the honest nodes. As [92] explained, the eventual delivery of messages can be realized by letting the sender repeat transmission until receiving an acknowledge from the receiver. However, when some receiver is faulty, this might cause an increasing buffer of outgoing messages, so we can let the sender only repeat transmissions of a limited number of outgoing messages. To preserve liveness in the handicapped network where each link only eventually delivers some messages (not all messages), we let each message carry a quorum certificate allowing its receiver sync up the latest progress.

4.3.2 Security goal

We aim at a secure asynchronous BFT consensus satisfying the following atomic broadcast (ABC) abstraction:

Definition 6. *In an atomic broadcast protocol among n nodes against f Byzantine corruptions, each node has an implicit input transaction* buf*, continuously selects some transactions from* buf *as actual input, and outputs a sequence of totally ordered transactions. In particular, it satisfies the following properties with all but negligible probability:*

- Agreement. *If one honest node outputs a tx, then every honest node outputs tx;*
- Total-order. *If any two honest nodes output sequences of transactions $\langle tx_0, tx_1, \ldots, tx_j\rangle$ and $\langle tx'_0, tx'_1, \ldots, tx'_{j'}\rangle$, respectively, then $tx_i = tx'_i$ for $i \leq min(j, j')$;*
- Liveness (strong validity [34] or censorship resilience). *If a transaction tx is input by any honest node, it will eventually output.*

In [34], Cachin *et al.* called the above liveness "*strong* validity", which recently was realized in DAG-rider [77] and DispersedLedger [124]. Strong validity is particularly useful for implementing state-machine replication API because it prevents censorship even if applying de-duplication techniques. Nevertheless, there are some weaker validity notions [34] ensuring a transaction to output only if all honest nodes (or $f + 1$ honest nodes) input it.

Note that there are also other complementary liveness notions orthogonal to validity, for example, [34] proposed "fairness" that means the relative confirmation latency of any two transactions is bounded (at least $f + 1$ honest nodes input them), and Kelkar et al. [78] and Zhang et al. [127] recently introduced "order-fairness". Nevertheless, following most studies about practical asynchronous BFT consensus, we only consider liveness in form of strong validity without fairness throughout the paper.

4.4 Dumbo-NG: Realizing Throughput-oblivious Latency

This section will elaborate our superior solution Dumbo-NG. We aim to support concurrent processes for bandwidth-intensive transaction dissemination and bandwidth-oblivious BA modules, so we can use much smaller batches to seize most bandwidth for realizing peak throughput. Here we elaborate our solution Dumbo-NG that implements the promising idea.

4.4.1 Overview of the Dumbo-NG protocol

At a very high level, Dumbo-NG consists of (i) n ever-running broadcasts and (ii) a sequence of BAs.

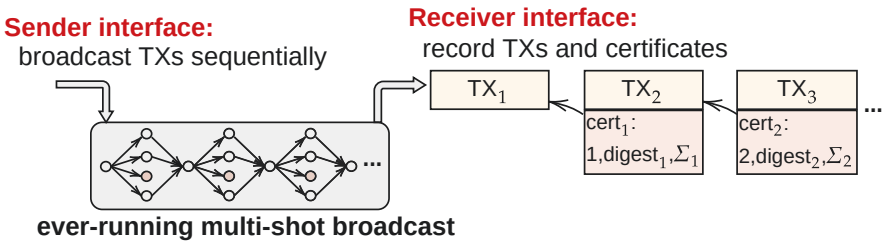


Figure 4.4: Ever-growing multi-shot broadcast.

Each node uses an ever-running multi-shot broadcast to continuously disseminate its input transactions to the whole network. As Figure 4.4 illustrates, the broadcast is never blocked to wait for any agreement modules or other nodes' broadcasts, and just proceeds by consecutive slots at its own speed. In each slot, the broadcast delivers a batch of transactions along with a quorum certificate (containing a threshold signature or concatenating enough digital signatures from distinct nodes). A valid certificate delivered in some slot can prove: at least $f + 1$ honest nodes have received the *same* transactions in all *previous* slots of the broadcast. The multi-shot broadcast is implementable, since each node only maintains several local variables (related to the current slot and immediate previous slot), and all earlier delivered transactions can be thrown into persistent storage to wait for the final output.

Because we carefully add certificates to the ever-running broadcasts, it becomes possible to concurrently execute MVBA with fine-tuned external validity condition to totally order the disseminated transactions. In particular, a node invokes an MVBA protocol, if $n - f$ distinct broadcasts deliver new transactions to it (so also deliver $n - f$ new certificates), and the node can take the $n - f$ certificates as MVBA input. The MVBA's external validity is specified to first check all certificates' validity and then check that these $n - f$ indeed correspond to some newly delivered transactions that were not agreed to output before. Once MVBA returns, all honest nodes receive a list of $n - f$ valid certificates, and pack the transactions certified by these certificates as a block of consensus output.

One might wonder that MVBA's external validity alone cannot ensures all broadcasted transactions are eventually output, because the adversary can let MVBA always return her input of $n - f$ certificates, which can always exclude the certificates of f honest nodes' broadcasts. As such, the adversary censors these f honest nodes. Nevertheless, the quality property of some recent MVBA protocols [9, 73, 88] can fortunately resolve the issue without incurring extra cost. Recall that quality ensures that with at least $1/2$ probability, the output of MVBA is from some honest node. So the probability of censorship decreases exponentially with the protocol execution.

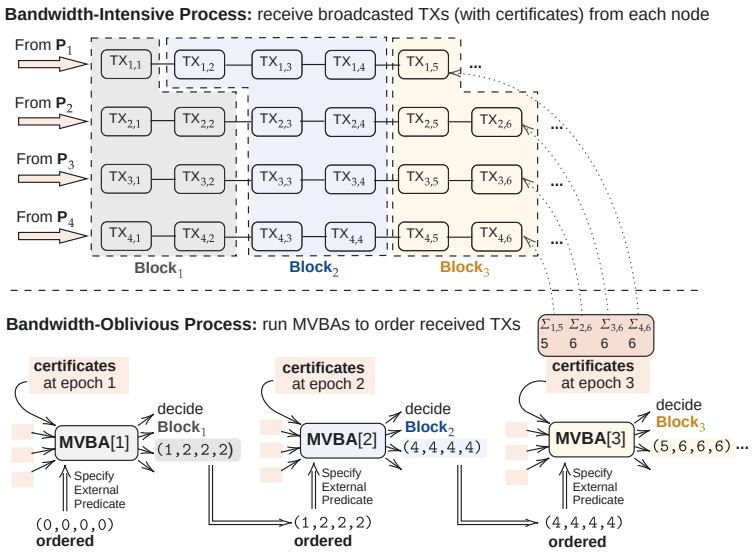


Figure 4.5: Illustration on how to totally order the received broadcasts through executing a sequence of MVBAs.

4.4.2 Details of the Dumbo-NG protocol

Here we present the formal description of Dumbo-NG. Algorithm 8 describes the main protocol of Dumbo-NG including two subprotocols for broadcasting transactions and ordering payloads. Algorithm 9 is about a daemon process Help and the function to call it. To better explain the algorithms, we list the local variables and give a brief description below.

- buf: A FIFO queue to buffer input transactions.
- $\mathcal{P}_j$: A designated node indexed by j.
- s: The slot number in a broadcast instance.
- e: The epoch tag of an MVBA instance.
- $\mathsf{TXs}_{j,s}$: The transaction batch received at the s-th slot of sender $\mathcal{P}_j$'s broadcast.
- $\mathsf{blocks}_{j,s}$: This is $\mathsf{TXs}_{j,s}$ thrown into persistent storage. It can be read and written by the broadcast process and/or the CallHelp process. MVBA process can also read it.
- block_e: The final consensus output decided at epoch e.
- ordered-indices: A vector to track how many slots were already placed into the final consensus output for each broadcast instance.
- $\sigma_{j,s}$: A partial threshold signature on $\mathsf{TXs}_{j,s}$.
- $\mathsf{ordered}_j$: The largest slot number for $\mathcal{P}_j$'s broadcast that has been ordered by consensus.
- $\mathsf{current}_j$: The current slot number of $\mathcal{P}_j$'s broadcast. This variable can be updated by the broadcast processes and is readable by the MVBA process.
- digest_j: A hash digest of transaction batch received from the sender $\mathcal{P}_j$ at slot $\mathsf{current}_j$. This is also readable by the MVBA process.
- Σ_j: The threshold signature for the transaction batch received from the sender $\mathcal{P}_j$ at slot $\mathsf{current}_j$, also readable by the MVBA process. We also call $(\mathsf{digest}_j, \Sigma_j)$ the broadcast quorum certificate.

- current-cert: A vector to store $(\mathsf{current}_j, \mathsf{digest}_j, \Sigma_j)$ for each $j \in [n]$.

As Figure 4.5 illustrates, Dumbo-NG is composed of two concurrent components, n ever-running multi-shot broadcasts and a sequence of MVBAs, which separately proceed as follows in a concurrent manner:

Broadcasts: There are n concurrent broadcasts with distinct senders. The sender part and receiver part of each broadcast proceed by slot s as follows, respectively:

- *Sender Part.* At the sender $\mathcal{P}_i$'s side, once it enters a slot s, it selects a $|B|$-sized batch $\mathsf{TXs}_{i,s}$ of transactions from buf, then multicasts it with the current slot index s (and probably a threshold signature Σ_{s-1} if $s > 1$) via PROPOSAL message, where Σ_{s-1} can be thought as a quorum certificate for the transaction batch $\mathsf{TXs}_{i,s-1}$ that was broadcasted in the preceding slot $s-1$. After that, the node $\mathcal{P}_i$ waits for $2f+1$ valid VOTE messages from distinct nodes. Since each VOTE message carries a threshold signature share for $\mathsf{TXs}_{j,s}$, $\mathcal{P}_i$ can compute a threshold signature Σ_s for $\mathsf{TXs}_{j,s}$. Then, move into slot $s+1$ and repeat.

- *Receiver Part.* At the receivers' side of a sender $\mathcal{P}_j$'s broadcast, if a receiving node $\mathcal{P}_i$ stays in slot s, it waits for a valid PROPOSAL message that carries $(s, \mathsf{TXs}_{j,s}, \Sigma_{j,s-1})$ from the designated sender $\mathcal{P}_j$. Then, $\mathcal{P}_i$ records $\mathsf{TXs}_{j,s}$, and also marks the transaction batch $\mathsf{TXs}_{j,s-1}$ received in the preceding slot $s-1$ as the "fixed" (denoted by $\mathsf{blocks}_j[s-1]$ and can be thrown into persistent storage). This is because $\Sigma_{j,s-1}$ attests that $\mathsf{TXs}_{j,s-1}$ was received and signed by enough honest nodes, so it can be fixed (as no other honest node can fix a different $\mathsf{TXs}'_{j,s-1}$). Meanwhile, $\mathcal{P}_i$ updates its local current-cert vector, by replacing the j-th element by $(s-1, \mathsf{digest}_{j,s-1}, \Sigma_{j,s-1})$, because $\mathcal{P}_i$ realizes the growth of $\mathcal{P}_j$'s broadcast. Next, $\mathcal{P}_i$ computes a partial signature σ_s on received proposal $\mathsf{TXs}_{j,s}$, and sends a VOTE message carrying σ_s to $\mathcal{P}_j$. Then $\mathcal{P}_i$ moves into the next slot $s+1$ and repeats the above. In case $\mathcal{P}_j$ (staying at slot s) receives a PROPOSAL message $(s', \mathsf{TXs}_{j,s'}, \Sigma_{j,s'-1})$ with some $s' > s$, it shall first retrieve missing transaction batches till slot $s'-1$ and then proceed in slot s' as above to vote on the latest received transaction

Algorithm 8 The Dumbo-NG protocol (for each node $\mathcal{P}_i$ runs the protocol consisting of the following processes:)

let buf to be a FIFO queue of input transactions, B to be the batch size parameter
Initializes current-cert := $[(\text{current}_1, \text{digest}_1, \Sigma_1), \ldots, (\text{current}_n, \text{digest}_n, \Sigma_n)]$ as $[(0, \perp, \perp), \ldots, (0, \perp, \perp)]$;
for every $j \in [n]$: $\text{blocks}_j \leftarrow \{\}$ (which shall be implemented by persistent storage)

/* **Broadcast-Sender** */
(one process that takes buf as input)

1: **for** each slot $s \in \{1, 2, 3, \ldots\}$ **do**
2: $\quad \text{TXs}_{i,s} \leftarrow$ buf[: B] to select a proposal, compute $\text{digest}_{i,s} \leftarrow \mathcal{H}(\text{TXs}_{i,s})$,
3: $\quad$ **if** $s > 1$: **multicast** PROPOSAL$(s, \text{TXs}_{i,s}, \text{digest}_{i,s-1}, \Sigma_{i,s-1})$, **else**: **multicast** PROPOSAL$(s, \text{TXs}_{i,s}, \perp, \perp)$
4: $\quad$ **wait** $2f + 1$ VOTE$(s, \sigma_{j,s})$ messages from $2f + 1$ distinct nodes $\{\mathcal{P}_j\}$ s.t. $\text{VrfyShare}_j(i \| s \| \text{digest}_{i,s}, \sigma_{j,s}) = \text{true}$
5: $\quad\quad$ compute the threshed signature Σ_s on $i \| s \| \text{digest}_{i,s}$ by combining $2f + 1$ received signature shares $\{\sigma_{j,s}\}_{j \in \{\mathcal{P}_j\}}$

/* **Broadcast-Receiver** */
(n processes that input and update current-cert and blocks_j)

6: **for** each $j \in [n]$: start a process to handle $\mathcal{P}_j$'s PROPOSAL messages as follows **do**
7: $\quad$ **for** each slot $s \in \{1, 2, 3, \ldots\}$ **do**
8: $\quad\quad$ **upon** receiving PROPOSAL$(s, \text{TXs}_{j,s}, \text{digest}_{j,s-1}, \Sigma_{j,s-1})$ message from $\mathcal{P}_j$ for the first time **do**
9: $\quad\quad\quad$ **if** $s = 1$ **then**
10: $\quad\quad\quad\quad \sigma_s \leftarrow \text{SignShare}_i(j \| 1 \| \mathcal{H}(\text{TXs}_{j,1}))$ to compute a partial sig on $\text{TXs}_{j,1}$
11: $\quad\quad\quad\quad$ record $\text{TXs}_{j,1}$, **send** VOTE$(1, \sigma_1)$ to $\mathcal{P}_j$
12: $\quad\quad\quad$ **if** $s > 1$ and $\text{digest}_{j,s-1} = \mathcal{H}(\text{TXs}_{j,s-1})$ and $\text{Vrfy}(j \| s-1 \| \text{digest}_{j,s-1}, \Sigma_{j,s-1}) = \text{true}$ **then**
13: $\quad\quad\quad\quad \text{blocks}_j[s-1] \leftarrow \text{TXs}_{j,s-1}$ to record the transaction proposal received in the precedent slot into persistent storage
14: $\quad\quad\quad\quad (\text{current}_j, \text{digest}_j, \Sigma_j) \leftarrow (s-1, \text{digest}_{j,s-1}, \Sigma_{j,s-1})$ to update the j-th element in the current-cert vector
15: $\quad\quad\quad\quad \sigma_s \leftarrow \text{SignShare}_i(j \| s \| \mathcal{H}(\text{TXs}_{j,s}))$ to compute partial sig on $\text{TXs}_{j,s}$
16: $\quad\quad\quad\quad$ record $\text{TXs}_{j,s}$ in memory and delete $\text{TXs}_{j,s-1}$ from memory, then **send** VOTE(s, σ_s) to $\mathcal{P}_j$
17: $\quad\quad$ **upon** receiving PROPOSAL$(s', \text{TXs}_{j,s'}, \text{digest}_{j,s'-1}, \Sigma_{j,s'-1})$ s.t. $s' > s$ from $\mathcal{P}_j$ for the first time **do**
18: $\quad\quad\quad$ **if** $\text{Vrfy}(j \| s'-1 \| \text{digest}_{j,s'-1}, \Sigma_{j,s'-1}) = \text{true}$ **then**
19: $\quad\quad\quad\quad$ send $\text{Pull}(j, s'-1, \text{digest}_{j,s'-1}, \Sigma_{j,s'-1})$ to its own CallHelp daemon (cf. Alg. 9)
20: $\quad\quad\quad\quad$ **wait for** $\text{blocks}_j[s-1], \ldots, \text{blocks}_j[s'-1]$ are all retrieved by the CallHelp daemon
21: $\quad\quad\quad\quad (\text{current}_j, \text{digest}_j, \Sigma_j) \leftarrow (s'-1, \text{digest}_{j,s'-1}, \Sigma_{j,s'-1})$ to update the j-th element in the current-cert vector
22: $\quad\quad\quad\quad \sigma_{s'} \leftarrow \text{SignShare}_i(j \| s' \| \mathcal{H}(\text{TXs}_{j,s'}))$ to compute the partial signature on $\text{TXs}_{j,s'}$
23: $\quad\quad\quad\quad$ record $\text{TXs}_{j,s'}$ in memory and delete $\text{TXs}_{j,s-1}$ from memory
24: $\quad\quad\quad\quad$ **send** VOTE$(s', \sigma_{s'})$ to $\mathcal{P}_j$, then move into slot $s \leftarrow s' + 1$

/* **Consensus for Ordering Payloads** */
(one process that inputs current-cert and blocks_j and outputs linearized blocks)

25: **initial** ordered-indices := $[\text{ordered}_1, \ldots, \text{ordered}_n]$ as $[0, \ldots, 0]$
26: **for** each epoch $e \in \{1, 2, 3, \ldots\}$ **do**
27: $\quad$ **initial** MVBA[e] with global predicate Q_e (to pick a valid current-cert′ with $n - f$ current_j increased w.r.t. ordered_j)
Precisely, the predicate Q_e is defined as: $Q_e(\text{current-cert}') \equiv$ (for each element $(\text{current}'_j, \text{digest}'_j, \Sigma'_j)$ of input vector current-cert′, $\text{Vrfy}(j \| \text{current}'_j \| \text{digest}'_j, \Sigma'_j) = \text{true}$ or $\text{current}'_j = 0) \wedge (\exists$ at least $n - f$ distinct $j \in [n]$, such that $\text{current}'_j > \text{ordered}_j) \wedge (\forall\ j \in [n]$, $\text{current}'_j \geq \text{ordered}_j)$
28: $\quad$ **wait** $\exists\ n - f$ distinct $j \in [n]$ s.t. $\text{current}_j > \text{ordered}_j$ ▷ This can be triggered by updates of current-cert
29: $\quad\quad$ input current-cert to MVBA[e]
30: $\quad\quad$ **wait** MVBA[e]'s output current-cert′ := $[(\text{current}'_1, \text{digest}'_1, \Sigma'_1), \ldots, (\text{current}'_n, \text{digest}'_n, \Sigma'_n)]$
31: $\quad\quad\quad \text{block}_e \leftarrow \text{sort}(\bigcup_{j \in [n]} \{\text{blocks}_j[\text{ordered}_j + 1], \ldots, \text{blocks}_j[\text{current}'_j]\})$, i.e. sort block_e canonically (e.g., lexicographically)
If some $\text{blocks}_j[k]$ to output was not recorded, send $\text{Pull}(j, \text{current}'_j, \text{digest}_{j,\text{current}'_j}, \Sigma_{j,\text{current}'_j})$ to its own CallHelp daemon to fetch the missing blocks from other nodes (because at least $f + 1$ honest nodes must record them), cf. Algorithm 9 for exemplary implementation of this function
32: $\quad\quad\quad$ buf $\leftarrow$ buf $\setminus$ block_e and **output** block_e, then **for** each $j \in [n]$: $\text{ordered}_j \leftarrow \text{current}'_j$

$\mathsf{TXs}_{j,s'}$ and then move to slot $s' + 1$. The details about how to pull transactions from other nodes will be soon explained in a later subsection. [4]

Agreements: Aside the transaction broadcasts, a separate asynchronous agreement module is concurrently executing and totally order the broadcasted transaction batches. The agreement module is a sequence of MVBAs and proceeds as follows by epoch e.

Each node initializes a vector (denoted by ordered-indices) as $[0, \ldots, 0]$ when $e = 1$. The j-th element in ordered-indices is denoted by $\mathsf{ordered}_j$ and represents how many transaction batches from the sender $\mathcal{P}_j$ have been totally ordered as output. Also, every node locally maintains a n-size vector denoted by current-cert to track the current progresses of all broadcasts. In particular, the j-th element $(\mathsf{current}_j, \mathsf{digest}_j, \Sigma_j)$ in current-cert tracks the progress of $\mathcal{P}_j$'s broadcast, and $\mathsf{current}_j$, digest_j and Σ_j presents the slot index, the hash digest, and the threshold signature associated to the last transaction batch received from the sender $\mathcal{P}_j$, respectively.

Then, a node waits for that at least $n - f$ broadcasts deliver new transactions (along with new certificates), i.e., $\mathsf{current}_j > \mathsf{ordered}_j$ for at least $n - f$ distinct j. Then, it invokes an MVBA[e] instance associated to the current epoch e with taking current-cert as input. The global predicate Q_e of MVBA[e] is fine-tuned to return a vector current-cert$' = [(\mathsf{current}'_1, \mathsf{digest}'_1, \Sigma'_1), \ldots, (\mathsf{current}'_n, \mathsf{digest}'_n, \Sigma'_n)]$, such that: (i) all Σ'_j is a valid threshold signature for the $\mathsf{current}'_j$-th slot of $\mathcal{P}_j$'s broadcast, and (ii) $\mathsf{current}'_j > \mathsf{ordered}_j$ for at least $n - f$ different $j \in [n]$. Finally, all nodes decide this epoch's output due to current-cert$'$. Specifically, they firstly check if $\mathsf{TXs}_{j,\mathsf{current}'_j}$ was received and $\mathsf{blocks}_j[\mathsf{current}'_j]$ was not recorded, if that is the case and $\mathsf{digest}'_j = \mathcal{H}(\mathsf{TXs}_{j,\mathsf{current}'_j})$, they mark $\mathsf{TXs}_{j,\mathsf{current}'_j}$ as fixed and record it as $\mathsf{blocks}_j[\mathsf{current}'_j]$. Then, the honest nodes pack the output of the epoch: for

[4] There is a subtle reason to first retrieve the missing transactions and then increase the local slot number in each broadcast instance, if a node is allowed to jump into a much higher slot without completing the pull of missing transactions, the asynchronous adversary might cause less than $f + 1$ honest nodes have the broadcasted transactions in its persistent storage. Our design ensures that a quorum certificate can certainly prove that $f + 1$ honest nodes indeed have thrown all previously broadcasted transactions (except the latest slot's) into their persistent storage (otherwise they wouldn't vote).

each $j \in [n]$, find the fixed transaction batches $\mathsf{blocks}_j[\mathsf{ordered}_j+1]$, $\mathsf{blocks}_j[\mathsf{ordered}_j+2]$, $\cdots$, $\mathsf{blocks}_j[\mathsf{current}'_j]$, and put these batches into the epoch's output. After output in the epoch e, each node updates ordered-indices by the latest indices in current-cert′, and enters the epoch $e+1$.

Algorithm 9 CallHelp daemon and Help daemon (for each node $\mathcal{P}_i$)

/* CallHelp **daemon** */

get access to the variables $[(\mathsf{current}_1, \mathsf{digest}_1, \Sigma_1), \ldots, (\mathsf{current}_n, \mathsf{digest}_n, \Sigma_n)]$ (which are initialized in Alg. 8)

This allows CallHelp pull missing transactions to sync up till the latest progress of each broadcast instance

initialize $\mathsf{max\text{-}missing}_j \leftarrow 0$, $\mathsf{max\text{-}missing\text{-}cert}_j \leftarrow \perp$ for each $j \in [n]$

upon receiving $\mathsf{Pull}(j, s^*, \mathsf{digest}_{j,s^*}, \Sigma_{j,s^*})$ **do** ▷ In case a few Pull messages are received,

  if $s^* > \mathsf{max\text{-}missing}_j$ and $s^* > \mathsf{current}_j$ and $\mathsf{Vrfy}(j \| s^* \| \mathsf{digest}_{j,s^*}, \Sigma_{j,s^*}) = \mathsf{true}$ **then**

    $\mathsf{max\text{-}missing}_j \leftarrow s^*$, $\mathsf{max\text{-}missing\text{-}digest}_j \leftarrow \mathsf{digest}_{j,s^*}$, $\mathsf{max\text{-}missing\text{-}cert}_j \leftarrow \Sigma_{j,s^*}$

    $\mathsf{missing}_j \leftarrow \mathsf{current}_j + 1$

    for $k \in \{\mathsf{missing}_j, \mathsf{missing}_j + 1, \mathsf{missing}_j + 2, ..., \mathsf{max\text{-}missing}_j\}$ **do**

If the CallHelp daemon receives more Pull messages for j-th broadcast while the loop is running, other Pull messages wouldn't trigger the loop but just probably update the break condition via $\mathsf{max\text{-}missing}_j$.

      if $k < \mathsf{max\text{-}missing}_j$ **then multicast** message $\textsc{CallHelp}(j, k)$;

      else $\textsc{CallHelp}(j, k, \mathsf{max\text{-}missing\text{-}cert}_j)$

      wait receiving $n - 2f$ valid $\mathsf{Help}(j, k, h, m_s, b_s)$ messages from distinct nodes

where "valid" means: for Help messages from the node $\mathcal{P}_s$, b_s is the valid s-th Merkle branch for Merkle root h and the Merkle tree leaf m_s

        interpolate $n - 2f$ received leaves $\{m_s\}$ to reconstruct and store $\mathsf{blocks}_j[k]$

/* Help **daemon** */

get access to read the persistently stored broadcasted tx blocks_j and the latest received tx $\mathsf{TXs}_{j,s}$ in Alg. 8 for each $j \in [n]$

upon receiving $\textsc{CallHelp}(j, k)$ or $\textsc{CallHelp}(j, k, \Sigma_{j,k})$ from node $\mathcal{P}_s$ for k for the first time **do**

  if $\mathsf{TXs}_{j,k}$ is the latest tx received and $\mathsf{Vrfy}(j \| k \| \mathcal{H}(\mathsf{TXs}_{j,k}), \Sigma_{j,k}) = \mathsf{true}$ **then**

    record $\mathsf{blocks}_j[k] \leftarrow \mathsf{TXs}_{j,k}$

  if $\mathsf{blocks}_j[k]$ is recorded **then**

    let $\{m_k\}_{k\in[n]}$ be fragments of $(n - 2f, n)$-erasure code applied to $\mathsf{blocks}_j[k]$, and h be Merkle tree root computed over $\{m_k\}_{k\in[n]}$

    send $\mathsf{Help}(j, k, h, m_i, b_i)$ to $\mathcal{P}_s$, where m_i is the i-th erasure-code fragment of $\mathsf{blocks}_j[k]$ and b_i is the i-th Merkle tree branch

Handle missing transaction batches: Note that is possible that some node might not store $\mathsf{blocks}_j[k]$, when (i) it has to put $\mathsf{blocks}_j[k]$ into its output after some MVBA returns

in epoch e or (ii) has to sync up to the k-th slot in the sender $\mathcal{P}_j$'s broadcast instance because of receiving a PROPOSAL message containing a slot number higher than its local slot. In both cases, each node can notify a CallHelp process to ask the missing transaction batches from other nodes, because at least $f + 1$ honest nodes must record or receive it because of the simple property of quorum certificate (otherwise they would not vote to form such certificates). We can adopt the techniques of erasure-code and Merkle tree used in verifiable information dispersal [38, 92] to prevent communication blow-up while pulling transactions. In particular, the CallHelp function is invoked to broadcast a CALLHELP message to announce that $\mathsf{blocks}_j[k]$ is needed. Once a node receives the CallHelp message, a daemon process Help (also cf. Algorithm 9) would be activated to proceed as: if the asked transaction batch $\mathsf{blocks}_j[k]$ was stored, then encode $\mathsf{blocks}_j[k]$ using an erasure code scheme, compute a Merkle tree committing the code fragments and i-th Merkle branch from the root to each i-th fragment. Along the way, the Help daemon sends the Merkle root, the i-th fragment, and the i-th Merkle branch to who is requesting $\mathsf{blocks}_j[k]$. Every honest node requesting $\mathsf{blocks}_j[k]$ can receive $f + 1$ valid responses from honest nodes with the same Merkle root, so it can recover the correct $\mathsf{blocks}_j[k]$. As such, each Help daemon only has to return a code fragment of the missing transactions under request, and the fragment's size is only $O(1/n)$ of the transactions, thus not blowing up the overall communication complexity.

4.4.3 Analyses of the Dumbo-NG protocol

Dumbo-NG realizes all requirements of ABC. The security intuitions are:

Safety intuitively stems from the following observations:

- *Safety of broadcasts*. For any sender $\mathcal{P}_j$'s broadcast, if a valid quorum certificate $\Sigma_{j,s}$ can be produced, at least $f + 1$ honest nodes have received the same sequence of transaction batches $\mathsf{TXs}_{j,s}$, $\mathsf{TXs}_{j,s-1}$, $\cdots$ $\mathsf{TXs}_{j,1}$. In addition, if two honest nodes locally store $\mathsf{blocks}_j[s]$ and $\mathsf{blocks}'_j[s]$ after seeing $\Sigma_{j,s}$ and $\Sigma'_{j,s}$, respectively, then $\mathsf{blocks}_j[s] = \mathsf{blocks}'_j[s]$. The above properties stem from the simple fact that quorum

certificates are $2f + 1$ threshold signatures on the hash digest of received transaction batches, so the violation of this property would either break the security of threshold signatures or the collision-resistance of cryptographic hash function.

- *External validity and agreement of* MVBA. The global predicate of every MVBA instance is set to check the validity of all broadcast certificates (i.e., verify threshold signatures). So MVBA must return a vector current-cert′ = [(current′$_1$, digest′$_1$, Σ'_1), ..., (current′$_n$, digest′$_n$, Σ'_n)], such that each (current′$_j$, digest′$_j$, Σ'_j) $\in$ current-cert′ is valid broadcast certificate. In addition, any two honest nodes would receive the same current-cert′ from every MVBA instance, so any two honest nodes would output the same transactions in every epoch, because each epoch's output is simply packing some fixed transaction batches according to current-cert′ returned from MVBA.

Liveness (censorship-resilience) is induced as the following facts:

- *Optimistic liveness of broadcasts*. If a broadcast's sender is honest, it can broadcast all input transactions to the whole network, such that all nodes can receive an ever-growing sequence of the sender's transactions with corresponding quorum certificates.

- *Quality of* MVBA. Considering that an honest sender broadcasts a transaction batch TXs$_{j,s}$ at slot s, all nodes must receive some quorum certificate containing an index equal or higher than s eventually after a constant number of asynchronous rounds. After some moment, all honest nodes would input such certificate to some MVBA instance. Recall the quality of MVBA, which states that with 1/2 probability, some honest node's input must become MVBA's output. So after expected 2 epochs, some honest node's input to MVBA would be returned, indicating that the broadcast's certificate with index s (or some larger index) would be used to pack TXs$_{j,s}$ into the final output.

- *Termination of* MVBA. Moreover, MVBA can terminate in expected constant asynchronous rounds. So every epoch only costs expected constant running time.

Now we formally prove the safety and liveness of Dumbo-NG in the presence of an asynchronous adversary that can corrupt $f < n/3$ nodes and control the network delivery.

Lemma 20. *If one honest node $\mathcal{P}_i$ records* $\mathsf{blocks}_k[s]$ *and another honest node $\mathcal{P}_j$ records* $\mathsf{blocks}_k[s]'$*, then* $\mathsf{blocks}_k[s] = \mathsf{blocks}_k[s]'$.

Proof: When $\mathcal{P}_i$ records $\mathsf{blocks}_k[s]$, according to the Algorithm 8, the node $\mathcal{P}_i$ has received a valid PROPOSAL$(s, \mathsf{TXs}_{k,s+1}, \Sigma_{k,s})$ message from $\mathcal{P}_k$ for the first time, where $\mathsf{Vrfy}(k||s||\mathcal{H}(\mathsf{TXs}_{k,s}), \Sigma_{k,s}) = \mathsf{true}$ and the $\Sigma_{k,s}$ is a threshold signature with threshold $2f + 1$. Due to the fact that each honest node only sends one VOTE message which carries a cryptographic threshold signature share for each slot s of $\mathcal{P}_k$, it is impossible to forge a threshold signature $\Sigma'_{k,s}$ satisfying $\mathsf{Vrfy}(k||s||\mathcal{H}(\mathsf{TXs}'_{k,s}), \Sigma'_{k,s}) = \mathsf{true}$ s.t. $\mathcal{H}(\mathsf{TXs}_{k,s}) \neq hash(\mathsf{TXs}'_{k,s})$. Hence, $\mathcal{H}(\mathsf{TXs}_{k,s}) = \mathcal{H}(\mathsf{TXs}'_{k,s})$, and following the collision-resistance of hash function, so if any two honest $\mathcal{P}_i$ and $\mathcal{P}_j$ records $\mathsf{blocks}_k[s]$ and $\mathsf{blocks}_k[s]'$ respectively, $\mathsf{blocks}_k[s] = \mathsf{blocks}_k[s]'$. □

Lemma 21. *Suppose at least $f + 1$ honest nodes record* $\mathsf{blocks}_k[s]$*, if node $\mathcal{P}_i$ does not record it and tries to fetch it via function* $\mathsf{CallHelp}(k, s)$*, then* $\mathsf{CallHelp}(k, s)$ *will return* $\mathsf{blocks}_k[s]$.

Proof: Since at least $f + 1$ honest nodes have recorded $\mathsf{blocks}_k[s]$, these honest nodes will do erasure coding in $\mathsf{blocks}_k[s]$ to generate $\{m_j^l\}_{j\in[n]}$, then compute the Merkle tree root h and the branch. Following the Lemma 20, any honest node who records $\mathsf{blocks}_k[s]$ has the same value, so it is impossible for $\mathcal{P}_i$ to receive $f + 1$ distinct valid leaves corresponds to another Merkle tree root $h' \neq h$. Hence, $\mathcal{P}_i$ can receive at least $f + 1$ distinct valid leaves which corresponds to root h. So after interpolating the $f + 1$ valid leaves, $\mathcal{P}_i$ can reconstruct $\mathsf{blocks}_k[s]$. □

Lemma 22. *If* $\mathsf{MVBA}[e]$ *outputs* $\mathsf{current\text{-}cert}' = [(\mathsf{current}'_1, \mathsf{digest}'_1, \Sigma'_1), \ldots, (\mathsf{current}'_n, \mathsf{digest}'_n, \Sigma'_n)]$*, then all honest nodes output the same* $\mathsf{block}_e = \bigcup_{j\in[n]}\{\mathsf{blocks}_j[\mathsf{ordered}_j + 1], \ldots, blocks_j[\mathsf{current}'_j]\}$.

Proof: According to the algorithm, all honest nodes initialize $\mathsf{ordered\text{-}indices} := [\mathsf{ordered}_1, \ldots, \mathsf{ordered}_n]$ as $[0, \ldots, 0]$. Then the $\mathsf{ordered\text{-}indices}$ will be updated by the output of MVBA, so following the agreement of MVBA, all honest nodes have the same $\mathsf{ordered\text{-}indices}$ vector when they participate in $\mathsf{MVBA}[e]$ instance. Again, following the

agreement of MVBA, all honest nodes have the same output from MVBA[e], so all of them will try to output $\mathsf{block}_e = \bigcup_{j\in[n]}\{\mathsf{blocks}_j[\mathsf{ordered}_j + 1], \ldots, \mathsf{blocks}_j[\mathsf{current}'_j]\}$.

For each $\mathsf{current}'_j$, if $(\mathsf{current}'_j, \mathsf{digest}'_j, \Sigma'_j)$ is a valid triple, i.e., $\mathsf{Vrfy}(j\|\mathsf{current}'_j\|\mathsf{digest}'_j, \Sigma_j) = \mathsf{true}$, then at least $f + 1$ honest nodes have received the $\mathsf{TXs}_{j,\mathsf{current}'_j}$ which satisfied $\mathsf{digest}'_j = \mathcal{H}(\mathsf{TXs}_{j,\mathsf{current}'_j})$. It also implies that at least $f + 1$ honest nodes can record $\mathsf{blocks}_j[\mathsf{current}'_j]$. By the code of algorithm, it is easy to see that at least $f + 1$ honest nodes have $\{\mathsf{blocks}_j[\mathsf{ordered}_j + 1], \ldots, \mathsf{blocks}_j[\mathsf{current}'_j]\}$. From Lemma 20, we know these honest nodes have same blocks. From Lemma 21, we know if some honest nodes who did not record some blocks wants to fetch it via CallHelp function, they also can get the same blocks. Hence, all honest nodes output same $\mathsf{block}_e = \bigcup_{j\in[n]}\{\mathsf{blocks}_j[\mathsf{ordered}_j + 1], \ldots, \mathsf{blocks}_j[\mathsf{current}'_j]\}$. □

Theorem 6. *The algorithm 8 satisfies* total-order*,* agreement *and* liveness *properties except with negligible probability.*

Proof: Here we prove the three properties one by one:

For agreement: Suppose that one honest node $\mathcal{P}_i$ outputs a $\mathsf{block}_e = \bigcup_{j\in[n]}\{\mathsf{blocks}_j[\mathsf{ordered}_j + 1], \ldots, \mathsf{blocks}_j[\mathsf{current}'_j]\}$, then according to the algorithm, the output of MVBA is $\mathsf{current\text{-}cert}' := [(\mathsf{current}'_1, \mathsf{digest}'_1, \Sigma'_1), \ldots, (\mathsf{current}'_n, \mathsf{digest}'_n, \Sigma'_n)]$.

Following Lemma 22, all honest nodes output the same $\mathsf{block}_e = \bigcup_{j\in[n]}\{\mathsf{blocks}_j[\mathsf{ordered}_j + 1], \ldots, \mathsf{blocks}_j[\mathsf{current}'_j]\}$. So the agreement is hold.

For Total-order: According to the algorithm, all honest nodes sequential participate in MVBA epoch by epoch, and in each MVBA, all honest nodes output the same block, so the total-order is trivially hold.

For Liveness: One honest node $\mathcal{P}_i$ can start a new broadcast and multicast his PROPOSAL message if it can receive $2f + 1$ valid VOTE messages from distinct nodes to generate a certificate. Note that the number of honest nodes is at least $n - f$ so sufficient VOTE messages can always be collected and $\mathcal{P}_i$ can start new multicast continuously. It also means that $\mathcal{P}_i$ would not get stuck.It also implies at least $n - f$ parallel broadcasts can grow

continuously since all honest nodes try to multicast their own PROPOSAL messages. Hence, each honest node can have a valid input of MVBA[e] which satisfies the predicate Q_e. In this case, we can immediately follow the termination of MVBA[e], then the MVBA[e] returns an output to all honest nodes.

According to the quality of MVBA, the input of honest nodes can be outputted with a probability no less than $1/2$, so even in the worst case, once the MVBA[e] decides to output the input of an honest node $\mathcal{P}_i$, the input of $\mathcal{P}_i$ will be outputted by all honest nodes. It also implies that the adversary can not censorship the input payloads from $\mathcal{P}_i$. Moreover, the probability of MVBA[e] outputting the input of node $\mathcal{P}_i$ is at most $O(1/n)$, so any honest input can be outputted in at most $O(n)$ asynchronous rounds in expected. □

Complexity and performance analysis. The round and communication complexities of Dumbo-NG can be analyzed as follows:

- The *round complexity* is expected *constant*. After a transaction is broadcasted by an honest node, every honest node would receive a valid quorum certificate on this transaction after 3 asynchronous rounds. Then, the transaction would output after expected two MVBA instances (due to the quality of MVBA). In case of facing faults and/or adversarial network, there could be more concrete rounds, for example, some nodes might need two rounds to retrieve missing transaction batches and MVBA could also become slower by a factor of 3/2.

- The amortized *communication complexity* is *linear* for sufficiently large batch size parameter. Due to our broadcast construction, it costs $O(|B|n + \lambda n)$ bits to broadcast a batch of $|B|$ transactions to all nodes. The expected communication complexity of an MVBA instance is $O(\lambda n^3)$. Recall that every MVBA causes to output a block containing at least $O(n|B|)$ transactions, and probably $O(n|B|)$ transactions need to bother Help and CALLHELP subroutines, costing at most $O(n^2|B| + \lambda n^3 \log n)$ bits. In sum, each epoch would output $O(n|B|)$ transactions with expected $O(n^2|B| + \lambda n^3 \log n)$ bits despite the adversary, which corresponds to linear amortized communication complexity if $|B| \geq \lambda n \log n$.

4.5 Implementation and Evaluations

We implement Dumbo-NG and deploy it over 16 different AWS regions across the globe. A series of experiments is conducted in the WAN settings with different system scales and input batch sizes. The experimental results demonstrate the superiority of Dumbo-NG over the existing performant asynchronous BFT consensus protocol Speeding-Dumbo (sDumbo) [73]. In particular, Dumbo-NG can preserve low latency (only several seconds) while realizing high throughput (100k tx/sec) at all system scales (from 4 to 64 nodes).

4.5.1 Implementation setup

Implementations details. We implement Dumbo-NG, sDumbo and Dumbo in Python3.[5] The same cryptographic libraries and security parameters are used throughout all implementations. The Dumbo-NG us implemented as two-process Python programs. Specifically, Dumbo-NG uses one process for broadcasting transactions and uses the other to execute MVBA. We use gevent library for concurrent tasks in one process. Coin flipping is implemented with using Boldyreva's pairing-based threshold signature [29]. Regarding quorum certificates, we implement them by concatenating ECDSA signatures. Same to HBBFT [92], Dumbo [74] and BEAT [58], our experiments focus on evaluating the performance of stand-alone asynchronous consensus, and all results are measured in a fair way without actual clients.

Implementation of asynchronous network. To realize reliable fully meshed asynchronous point-to-point channels, we implement a (persistent) unauthenticated TCP connection between every two nodes. The network layer runs on two separate processes: one handles message receiving, and the other handles message sending. If a TCP connection is dropped (and fails to deliver messages), our implementation would attempt to re-connect.

[5]Proof-of-concept implementation is available at https://github.com/fascy/Dumbo_NG. Though our proof-of-concept implementation didn't implement the processes for pulling missing transactions in Dumbo-NG, it cautiously counts the number of such retrievals and found that there were less than 1% missing transaction batches to retrieve in all WAN evaluations.

Setup on Amazon EC2. We run Dumbo-NG, Dumbo and Speeding-Dumbo (sDumbo) among EC2 c5.large instances which are equipped with 2 vCPUs and 4 GB main memory. Their performances are evaluated with varying scales at $n = 4, 16$, and 64 nodes. Each transaction is 250-byte to approximate the size of a typical Bitcoin transaction with one input and two outputs. For $n = 16$ and 64, all instances are evenly distributed in 16 regions across five continents: Virginia, Ohio, California, Oregon, Canada, Mumbai, Seoul, Singapore, Sydney, Tokyo, Frankfurt, London, Ireland, Paris, Stockholm and São Paulo; for $n = 4$, we use the regions in Virginia, Sydney, Tokyo and Ireland.

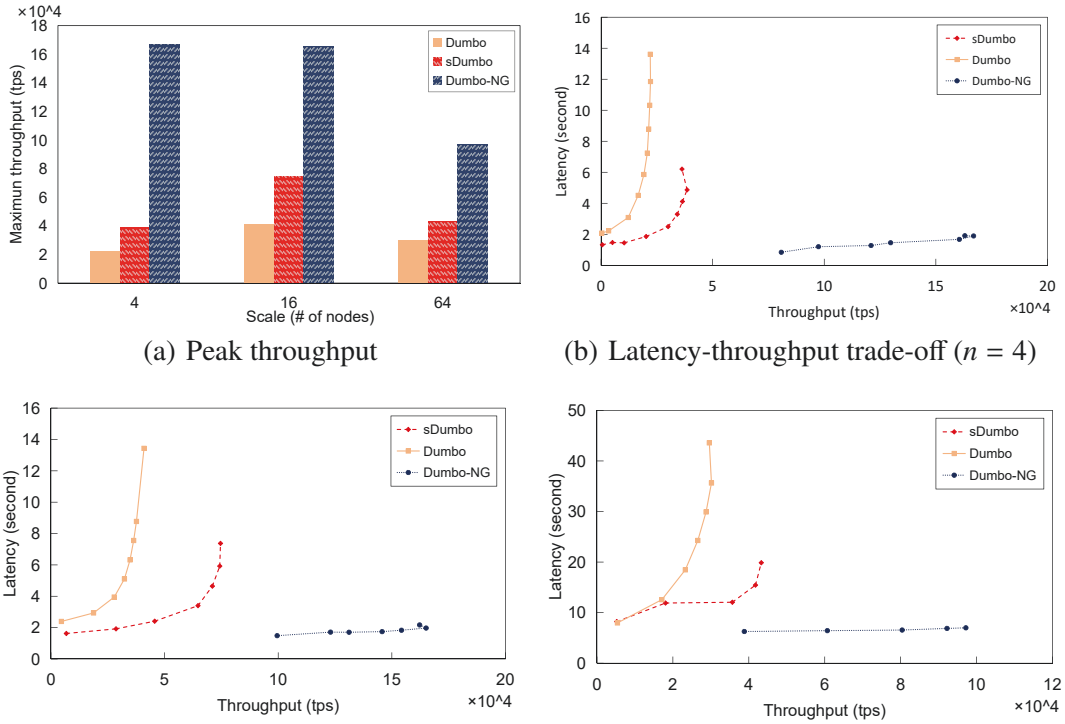
4.5.2 Evaluations in the WAN setting

Dumbo-NG **v.s. prior art**. To demonstrate the superior performance of Dumbo-NG, we first comprehensively compare it to sDumbo and Dumbo. Specifically,

- *Peak throughput.* For each asynchronous consensus, we measure its throughput, i.e., the number of transactions output per second. The peaks of the throughputs of Dumbo-NG, sDumbo and Dumbo (in varying scales) are presented in Figure 4.6(a). This reflects how well each protocol can handle the application scenarios favoring throughput. Overall, the peak throughput of Dumbo-NG has a several-times improvement than any other protocol. Specifically, the peak throughput of Dumbo-NG is more than 7x of Dumbo when $n = 4$, about 4x of Dumbo when $n = 16$, and roughly 3x of Dumbo when $n = 64$. As for sDumbo, it is around 4x of sDumbo when $n = 4$, over 2x of sDumbo when $n = 16$, and almost 3x of sDumbo when $n = 64$.

- *Latency-throughput trade-off.* Figure 4.6(b), 4.6(c) and 4.6(d) illustrate the latency-throughput trade-off of Dumbo-NG, sDumbo and Dumbo when $n = 4$, 16 and 64, respectively. Here *latency* is the time elapsed between the moment when a transaction appears in the front of a node's input buffer and the moment when it outputs, so it means the "consensus latency" excluding the time of queuing in mempool. The trade-off between latency and throughput determines whether a BFT protocol can simultaneously handle throughput-critical and latency-critical applications. To measure Dumbo-NG's

(average) latency, we attach timestamp to every broadcasted transaction batch, so all nodes can track the broadcasting time of all transactions to calculate latency. The experimental results show: although Dumbo-NG uses a Byzantine agreement module same to that in sDumbo (i.e., the GLL+22-MVBA), its trade-off surpasses sDumbo in all cases. The more significant result is that at all system scales, Dumbo-NG preserves a low and relatively stable latency (only a few seconds), while realizing high throughput at the magnitude of 100k tx/sec. In contrast, other protocols suffer from dramatic latency increment while approaching their peak throughput. This demonstrates that Dumbo-NG enjoys a much broader array of application scenarios than the prior art, disregarding throughput-favoring or latency-favoring.



(a) Peak throughput

(b) Latency-throughput trade-off ($n = 4$)

(c) Latency-throughput trade-off ($n = 16$)

(d) Latency-throughput trade-off ($n = 64$)

Figure 4.6: Performance of Dumbo-NG in comparison with the state-of-the-art asynchronous protocols (in the WAN setting).

Latency/throughput while varying batch sizes. The tested asynchronous protocols do have a parameter of batch size to specify that each node can broadcast up to how many transactions each time. As aforementioned, the latency-throughput tension in many earlier

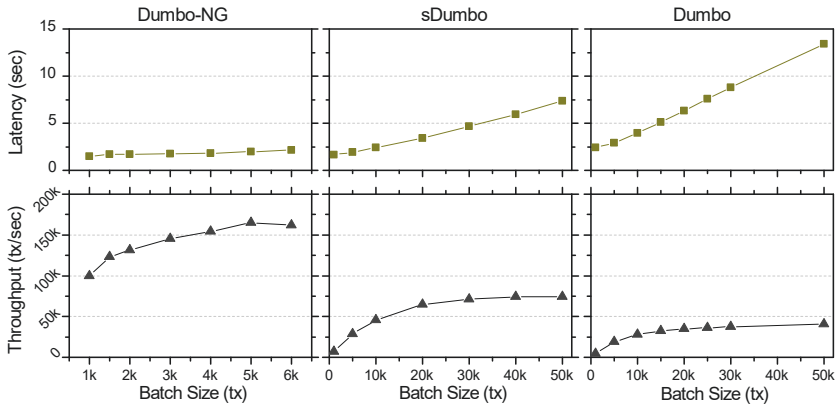


Figure 4.7: Throughput/latency of Dumbo-NG, sDumbo and Dumbo in varying batch size for WAN setting ($n = 16$).

practical asynchronous BFT consensuses like HBBFT and Dumbo is actually related to the choice of batch size: their batch size parameter has to be tuned up for higher throughput, while this might cause a dramatic increment in latency.

Here, we gradually increase the batch size and record the latency and throughput to see how batch size takes effect in the tested protocols. Figure 4.7 plots a sample when $n = 16$ for Dumbo-NG, sDumbo and Dumbo. It is observed that with the increment of batch size, the throughput of all protocols starts to grow rapidly but soon tends to grow slowly. The latency of Dumbo and sDumbo grows at a steady rate as the batch size increases. However, the latency of Dumbo-NG remains constantly small. When the batch size increases from 1k to 5k, Dumbo-NG reaches its peak throughput (about 160k tx/sec), and its latency only increases by less than 0.5 sec; in contrast, sDumbo and Dumbo have to trade a few seconds in their latency for reaching peak throughput. Clearly, Dumbo-NG needs a much smaller batch size (only about 1/10 of others) to realize the highest throughput, which allows it to maintain a pretty low latency under high throughput.

4.5.3 More evaluations in the controlled delay/bandwidth settings

The above experiments in the WAN setting raises an interesting question about Dumbo-NG: *why it preserves a nearly constant latency despite throughput?* We infer the following two conjectures based on the earlier WAN setting results:

1. The MVBA protocols are actually insensitive to the amount of available bandwidth, so no matter how much bandwidth is seized by transaction broadcasts, their latency would not change as only rely on round-trip time of network.

2. The nodes in Dumbo-NG only need to broadcast a small batch of transactions (e.g., a few thousands that is 1/10 of Dumbo) to closely track bandwidth.

Clearly, if the above conjectures are true, the latency of Dumbo-NG would just be 1-2x of MVBA's running time. Hence, we further conduct extensive experiments in a LAN setting (consisting of servers in a single AWS region) with manually controlled network propagation delay and bandwidth to verify the two conjectures, respectively.

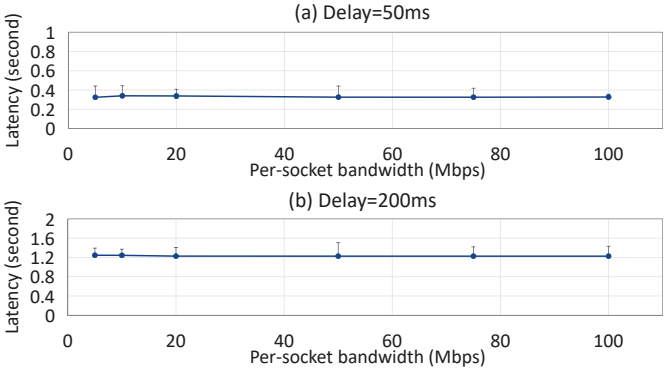


Figure 4.8: Latency of GLL+22-MVBA [73] with n=16 nodes in varying bandwidth for (a) 50ms and (b) 200ms one-way network delay, respectively.

Evaluate MVBA with controlled network bandwidth/delay. We measure MVBA, in particular, its latency, in the controlled experiment environment (for n=16 nodes). Here the input-size of MVBA is set to capture the length of n quorum certificates. Figure 4.8 (a) and (b) show the results in the setting of 50 and 100 ms network propagation delays, respectively, with varying the bandwidth of each peer-to-peer tcp link (5, 10, 20, 50, 75, or 100 Mbps). Clearly, our first conjecture is true, as MVBA is definitely bandwidth-oblivious, as its latency relies on propagation delay other than available bandwidth.

Test Dumbo-NG with controlled network bandwidth/delay. Then we verify whether Dumbo-NG can closely track available bandwidth resources with only small batch sizes. We examine how Dumbo-NG gradually saturates bandwidth resources whiling batch size

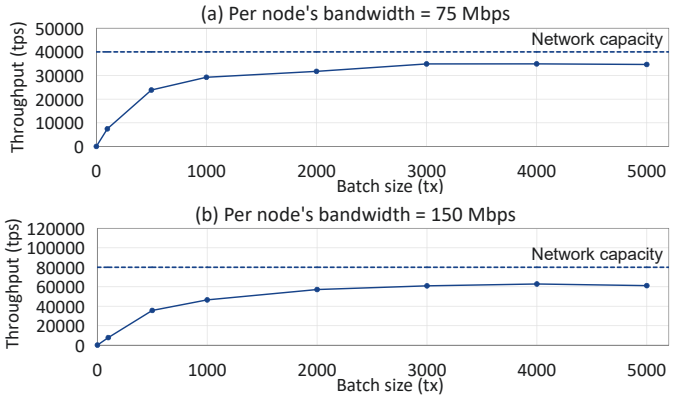


Figure 4.9: The dependency of throughput on varying batch size in controlled deployment environment with 50 ms one-way delay and (a) 75Mbps and (b) 150Mbps bandwidth.

increases, in a controlled environment (for n=16 nodes). The one-way network delay is set to 50ms, and per-node's bandwidth is set for (a) 75Mbps (5Mbps per tcp socket) or (b) 150Mbps (10Mbps per tcp socket). The controlled network parameters well reflect an inter-continental communication network. Clearly, the results verify our second conjecture that Dumbo-NG can fully utilize bandwidth while using small batch sizes (less than 1MB).

Behind the throughput-oblivious latency. Given the extensive experiment results, we now can understand why the latency of Dumbo-NG is almost independent to its throughput: (i) some small batch sizes can already fully utilize network bandwidth, and therefore the latency of broadcasting a batch transactions is much smaller than that of MVBA; (ii) when the broadcast instances seize most bandwidth, the latency of MVBA would not be impacted as its latency is bandwidth-insensitive. So if a broadcast makes a progress when the MVBA of epoch e is running, this progress would be solicited to output by the MVBA of next epoch $e + 1$ (if no fault), and even if there are $n/3$ faulty nodes, it is still expected to output by the MVBA of epoch $e + 2$ (due to MVBA's quality), which results in throughput-oblivious latency. For sake of completeness, we also interpret the above intuition into an analytic formula and perform numerical analysis to translate it into a quantitative study as follows: Assume that all nodes have the equal bandwidth w and all p2p links have the same round-trip delay τ, and we might ignore some constant coefficients in formulas.

For Dumbo-NG, its throughput/latency can be roughly written:

$$\text{tps of Dumbo-NG} = \frac{nB}{nB/w + \tau}, \text{ latency of Dumbo-NG} = nB/w + \tau + 1.5 \cdot T_{BA}$$

where $(nB/w + \tau)$ reflects the duration of each broadcast slot, and nB represents the number of transactions disseminated by all n nodes in a slot. Recall that our experiments in Section 4.5 demonstrate that the agreement modules are bandwidth-oblivious and cost little bandwidth, so we ignore the bandwidth used by the agreement modules in Dumbo-NG. Hence, the term nB/w reflects the time to disseminate B transactions to all nodes while fully utilizing w bandwidth, and τ is for the round-trip delay waiting for $n - f$ signatures to move in the next slot. The term T_{BA} represents the latency of MVBA module, and the factor 1.5 captures that a broadcast slot might finish in the middle of an MVBA execution and on average would wait 0.5 MVBA to be solicited by the next MVBA's input.

For Dumbo/HBBFT, the rough throughput/latency formulas are:

$$\text{tps of HBBFT variants} = \frac{nB}{nB/w + \tau + T_{\text{BA}} + T_{TPKE}} < \frac{nB}{nB/w + T_{\text{BA}}}$$

$$\text{latency of HBBFT variants} = \frac{nB}{w} + \tau + T_{\text{BA}} + T_{TPKE} > nB/w + T_{\text{BA}}$$

where $nB/w + T_{\text{BA}}$ represents the duration of each ACS, and nB reflects the number of transactions that are output by every ACS (here we ignore some constant communication blow-up factor, so would we do in the following analysis). The term nB/w captures the time to disseminate B transactions, T_{BA} denotes the latency of running the Byzantine agreement phase (e.g., one MVBA in Dumbo or n ABAs in HBBFT), T_{TPKE} represents the delay of threshold decryption/encryption for preventing censorship, and τ reflects the network propagation delay involved in the phase of transaction dissemination. To simplify the formulas, we might omit τ and T_{TPKE}, which still allows us to estimate the upper bound of Dumbo/HBBFT's throughput and the lower bound of their latency.

Noticeably, both throughput formulas have a limit close to network bandwidth w, but their major difference is whether the T_{BA} term appears at the denominator of throughput or not (representing whether the Byzantine agreement module blocks transaction dissemination or not). We specify parameters to numerically analyze this impact on throughput-latency trade-off. In particular, for n=16, we set the per-node bandwidth w as 150 Mbps, round-trip delay τ as 100 ms, the latency of Byzantine agreement T_{BA} as 1 second, and transaction size as 250 bytes. The throughput-latency trade-offs induced from the above formulas are plotted in Figure 4.10 (where the throughput varies from 20% of to 90% of network capacity).

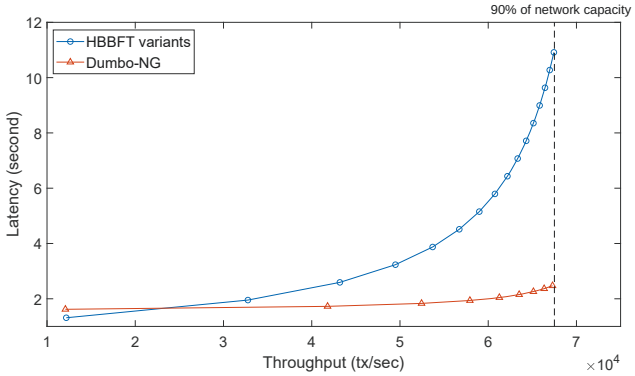


Figure 4.10: Numerical analysis to show the throughput-latency trade-offs in Dumbo-NG and HBBFT variants.

Clearly, despite their same throughput limitation, the two types of protocols present quite different throughput-latency trade-offs. In particular, when their throughputs increase from the minimum to 90% of network capacity, the latency increment of the Dumbo-NG is only 0.85 sec (only ~50% increment), while Dumbo suffers from 9.60 second increment (~630% increment). This reflects that Dumbo-NG can seize most network bandwidth resources with only small batch sizes, because its transaction dissemination is not blocked by the slow Byzantine agreement modules.

4.6 Discussions

Flooding launched by malicious nodes. In the HBBFT and DAG type of protocols [54, 58, 73, 74, 77, 92], the malicious nodes cannot broadcast transactions too much faster

than the honest nodes, because all nodes explicitly block themselves to wait for the completeness of $n - f$ broadcasts to move forward. While one might wonder that in Dumbo-NG, the malicious nodes probably can broadcast a huge amount of transactions in a short term, which might exhaust the resources of the honest nodes and prevent them from processing other transactions.

Nevertheless, this is actually a general flooding attack in many distributed systems, and it is not a particularly serious worry in Dumbo-NG, because a multitude of techniques already exist to deter it. For example, the nodes can allocate an limited amount of resources to handle each sender's broadcast, so they always have sufficient resources to process the transactions from the other honest senders, or an alternative mitigation can also be charging fees for transactions as in Avalanche [114].

Input tx buffer assumptions related to censorship-resilience. Censorship-resilience (liveness) in many work [58, 73, 74, 92] explicitly admits an assumption about input buffer (a.k.a. backlog or transaction pool): a transaction is guaranteed to eventually output, only if it has been placed in all honest nodes' input buffer. Our censorship resilience allows us to adopt a different and weaker assumption about the input buffer (that is same to DispersedLedger [124], Aleph [70] and DAG-rider [77]): if a transaction appears in any honest node's input buffer (resp. k random nodes' input buffers), the transaction would output eventually (resp. output with all but negligible probability in k).

Our input buffer assumption is appropriate or even arguably quintessential in practice. First, in many consortium blockchain settings, a user might be allowed to contact only several consensus nodes. For example, a Chase bank user likely cannot submit her transactions to a consensus node of Citi bank. Moreover, even if in a more open setting where a client is allowed to contact all nodes, it still prefers to fully leverage the strength of our censorship resilience property to let only k consensus nodes (instead of all) to process each transaction for saving communication cost.

4.6.1 From validity to censorship resilience.

The validity of asynchronous atomic broadcast captures that a certain input transaction would eventually output. It has a few fine-grained flavors [34]:

- *Strong validity* (called censorship resilience throughout the paper): tx can eventually output, if *any* honest node takes it as input;
- *Validity*: if $f + 1$ honest nodes input tx, it can eventually output;
- *Weak validity*: if *all* honest nodes input tx, it eventually outputs.

The following examples illustrate why strong validity can easily prevent censorship of transactions when constructing SMR from atomic broadcast, whereas weaker flavors of validity cannot:

- *In permissioned settings*: Strong validity has practical meaning in real-world consortium blockchain systems, because a client might not be allowed to contact all consensus nodes, and can only rely on several designated nodes to process its transactions [124]. In this setting, strong validity is critical because only it ensures that every client can have the input transactions to eventually output as long as the client has permission to contact an honest consensus node.
- *Enable de-duplication*: In more open settings where a client has the permission to contact all nodes to duplicate its transactions, strong validity is still important as it empowers de-duplication techniques [54, 121] to reduce redundant communication. For example, each transaction can be sent to only k nodes (where k is a security parameter), because the k random nodes would contain at least one honest node with a probability exponentially large in k (in case of static corruptions); in contrast, a client has to contact $f + 1$ honest nodes (resp. $2f + 1$ honest nodes) if there is only validity (resp. weak validity).

One might wonder whether quality [65] alone can prevent censorship in the asynchronous setting. Recall that in many asynchronous protocols with quality but without strong validity (e.g., asynchronous common subset [23] and Tusk [54]), the adversary can indeed

drop f honest nodes. Thus, quality only implies liveness *conditioned on* that all honest nodes input the same transactions redundantly. In other words, if aiming at liveness from quality directly, it needs to diffuse transactions over all honest nodes. This "approach", unfortunately, might incur $O(n)$ communication blow-up as discussed in Section 1. Alternatively, quality together with threshold encryption can prevent censorship [92] but is computationally costly.

4.6.2 Tips on production-level implementation

Challenges and tips to production-level implementation. For production-level implementation of Dumbo-NG with bounded memory, a few attentions (some of which are even subtle) need to be paid. First, the MVBA instantiation shall allow nodes to halt after they decide output (without hurting other nodes' termination), such that a node can quit old MVBA instances and then completely clean them from memory. Also, similar to Tusk [54], messages shall carry latest quorum certificates to attest that $f + 1$ honest nodes have stored data in their persistent storage, such that when a slow node receives some "future" messages, it does not have to buffer the future messages and can directly notify a daemon process to pull the missing outputs accordingly. In addition, Dumbo-NG has a few concurrent tasks (e.g., broadcasts and MVBAs) that rely on shared global variables. This is not an issue when implementing these tasks by multiple threads in one process. Nevertheless, when separating these tasks into multiple processes, inter-process communication (IPC) implementation has to correctly clean IPC buffers to avoid their memory leak due to long network delay. Last but not least, if a node constantly fails to send a message to some slow/crashed node, it might keeps on re-sending, and thus its out-going message buffer might dramatically increase because more and more out-going messages are queued to wait for sending. It can adopt the practical alternative introduced by Tusk, namely, stop (re-)sending too old out-going messages and clean them from memory, because messages in Dumbo-NG can also embed latest quorum certificates (to help slow nodes sync up without waiting for all protocol messages).

Here we extend the discussions to give more detailed suggestions for practitioners.

1. *Halt in asynchronous* BA *without hurting termination*. In many asynchronous BA protocols [9, 35], the honest nodes cannot simultaneously decide their output in the same iteration. So even if a node has decided its output, it might need to continue the execution to help other nodes also output (otherwise, there might exist a few nodes fail to output). Fortunately, a few studies [26,74,89,99] demonstrated how to securely halt in asynchronous BAs without hurting termination. Our MVBA instantiation [73] also has the feature of immediate halt after output, as the honest nodes would always multicast a quorum certificate proving the decided output and then quit.

2. *Share variables across concurrent processes*. Some global variables are shared among the concurrent tasks in Dumbo-NG, for example, the task of MVBAs n shall have access to read the latest broadcast certificates generated in the tasks of n running broadcasts. When practitioners implement these tasks with different processes, inter-process communication (IPC) for sharing these global variables shall be cautiously handled. For example, if IPC socket is used to pass the latest broadcast certificates to the MVBAs' process, it is important to implement a thread that executes concurrent to MVBAs to continuously take certificates out of the IPC sockets and only track the latest certificates (otherwise, a huge number of certificates could be accumulated in the receiving buffer of IPC socket if a single MVBA is delayed).

3. *Use quorum certificates for retrievability to help slow nodes*. Up to f slow honest nodes might receive a burst of "future" messages in an asynchronous network. There are three such cases: (i) a broadcast sender receives Vote messages higher than its local slot; (ii) a broadcast receiver gets proposal messages higher than its local slot; or (iii) any nodes receivers some MVBA messages with epoch number higher than its local epoch. For case (i), it is trivial that the broadcast sender can just omit such "future" Vote messages, because these messages must be sent by corrupted nodes. For case (ii), we have elaborated the solution in Section 4.4: when a slow node staying at slot s receives a proposal message with valid quorum certificate but slot $s' > s$, it needs to first pull the missing transactions till slot $s' - 1$ and then increase

its local slot number to continue voting in slot s' and later slots. For case (iii), it can also be trivially solved by letting the e-th epoch's MVBA messages carry the unforgeable quorum certificate for MVBA[$e-1$]'s output, such that upon a slow node receives some MVBA message belong to a "future" epoch e' larger than its local epoch e, it can pull all missing MVBA outputs till epoch $e'-1$ (similar to pull broadcasted transactions) and then move into epoch e'.

4. *Avoid infinite buffer of out-going messages*. To correctly implement asynchronous communication channels, a message sending node might continuously re-send each protocol message until the message receiving node returns an acknowledgment receipt (e.g., through TCP connection). As such, the message sending node might have more and more out-going messages accumulated while stucking in re-sending some very old messages. At first glance, it seemingly requires infinite memory to buffer these out-going messages. However, recall that Tusk [54] can stop re-sending old out-going messages and then securely clean them. The implementation of Dumbo-NG can also adapt the idea to bound the size of out-going buffer by cleaning the "old" out-going messages belong to slot/epoch smaller than the current local slot/epoch, as long as practitioners follow the guidance in (3) to embed previous slot/epoch's quorum certificate in the current slot/epoch's out-going messages to help slow nodes to pull missing outputs by a quorum certificate (instead of actually receiving all sent protocol messages).

4.7 Summary

In this chapter, we present Dumbo-NG—a direct, concise and efficient reduction from asynchronous BFT atomic broadcast to multi-valued validated Byzantine agreement (MVBA). In greater detail, the core contributions of Dumbo-NG can be summarized as follows:

- *Resolve the latency-throughput tension*. Dumbo-NG resolves the severe tension between throughput and latency in HBBFT/Dumbo. Recall the issue stems from:

for higher throughput, the broadcasts in HBBFT/Dumbo have to sacrifice latency to disseminate a huge batch of transactions, and this is needed to "contend" with the agreement modules to seize more bandwidth resources.

Dumbo-NG solves the issue and can approach the maximum throughput without trading latency, i.e., realize *throughput-oblivious latency*. This is because it supports to run the bandwidth-intensive transaction broadcasts completely concurrently to the bandwidth-oblivious agreement modules. Remark that the concurrent execution of broadcasts and agreement is non-trivial in the asynchronous setting, as we need carefully propose and implement a few properties of broadcast and agreement to bound communication complexity and ensure censorship resilience.

Table 4.1: Validity (liveness) of asynchronous atomic broadcast if stressing on nearly *linear* amortized communication

	Strong validity (Definition 4.1)?	Memory-bounded implementation?
DAG-Rider [77], DispersedLedger [124], and Aleph [70]	✓∗	○†
Tusk [54]	✗suboptimal comm.; or after GST‡	✓
HBBFT [92], Dumbo [74], and variants [58, 73]	✓diffuse TX+TPKE for de-duplication	✓
Dumbo-NG (this paper)	✓∗	✓

∗ Here we assume de-duplicated input buffers in DAG-Rider, DispersedLedger and Dumbo-NG, which can be realized (i) in a permissioned setting where a client only has permission to contact several nodes or (ii) by de-duplication techniques [51, 54, 121].
† The memory-bounded implementation of [70, 77, 124] is unclear.
‡ Though Tusk employs transaction de-duplication techniques to send transactions to only k nodes, it doesn't realize strong validity, so still needs $k = f + 1$ to ensure all transactions to output in the worst-case asynchronous network; and a recent improvement of Tusk— Bullshark [69] presents an implementation that explicitly stresses on strong validity only after global stabilization time (GST).

- *Prevent censorship with minimal cost.* As shown in Table 4.1, similar to DAG-Rider [77] and DispersedLedger [124], Dumbo-NG ensures that any transaction input by an honest node can eventually output (a.k.a. *strong* validity in [34]), and thus when building a state-machine replication (SMR) service [117] from such atomic

broadcasts, one can expect to overcome potential censorship with minimized extra cost (e.g., by directly using de-duplication techniques [54, 121]). So we call strong validity and censorship resilience interchangeably, and can safely assume the honest parties have de-duplicated input buffers containing mostly different transactions throughout the paper. Such resilience of censorship is born with our new protocol structure, because no matter how slow a broadcast can be, the concurrently running agreement modules can eventually pick it into the final output through a quorum certificate pointing to it. As such, Dumbo-NG does not rely on additional heavy cryptographic operations (e.g, [58, 73, 74]) or sub-optimal redundant communication (e.g., the worst case of Tusk [54]) to realize guaranteed resistance against an asynchronous censorship adversary. This further demonstrates the strength of our result w.r.t its security aspect in addition to its practicality.

- *Implementation and extensive experiments over the Internet.* We also implement Dumbo-NG and extensively test it among $n = 4$, 16 or 64 nodes over the global Internet, with making detailed comparison to the state-of-the-art asynchronous protocols including Dumbo and sDumbo [73]. At all system scales, the peak throughput of Dumbo-NG is multiple times better than any of the other tested asynchronous BFT protocols, e.g., 4-8x over Dumbo and 2-4x over sDumbo. More importantly, the latency of Dumbo-NG is significantly less than others (e.g., 4-7x faster than Dumbo when both protocols realizes the maximum throughput). Actually, the latency of Dumbo-NG is nearly independent to its throughput, indicating how effective it is to resolve the throughput-latency tension lying in the prior designs.

CHAPTER 5

BOLT-DUMBO TRANSFORMER: ASYNCHRONOUS CONSENSUS AS FAST AS THE PIPELINED BFT

This chapter presents Bolt-Dumbo Transformer, the first generic and practical framework for optimistic asynchronous atomic broadcast, whose performance is proportional to network conditions. Specifically, it achieves the same performance as the deterministic BFT consensus on the normal Internet and also keeps the same performance as the asynchronous BFT consensus when the network changes to be asynchronous.

5.1 Background

The explosive popularity of decentralization [32, 100] 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 [39]. 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 the output of honest parties.

5.1.1 Motivation

A robust BFT is desirable in the absence of synchrony. Most practical BFT protocols were studied under well-connected network conditions. For example, classic synchrony assumption needs all messages to deliver within a known delay, and its weaker variant called partial synchrony [59] (a.k.a. eventual synchrony) assumes that after an unknown global stabilization time (GST), all messages can be delivered synchronously. In the

wide-area network, these assumptions about synchrony may not always hold. What's worse, in an asynchronous network [14], such (partially) synchronous protocols [11, 12, 15, 16, 24, 41, 44, 71, 72, 105, 122] will grind to a halt (i.e., suffers from the inherent *loss of liveness* [60, 92]), and Bitcoin might even have a *safety* issue of potential double-spending [115] 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 [34, 58, 74, 92, 124] 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" [60] 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 [92] and Dumbo-BFT [74] (and their latest improved variants [73, 124]) 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 [125][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 [74]. Even

[1]Remark that [125] 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.

worse, the inferior latency performance of asynchronous protocols stems from the fact that all parties generate some common randomness (e.g., "common coin" [35, 40]), 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 [73], 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 [10], e.g., 5 in the two-chain HotStuff and 3 in PBFT.

Hence, BFT consensus protocols for the open Internet face a basic design "dilemma": 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?

5.1.2 Challenges

Efficiency obstacles in prior art. As briefly mentioned, pioneering works of Kursawe-Shoup [80] (KS02) and a following improvement of Ramasamy-Cachin [113] (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. As Fig. 5.1 illustrates, 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 [61].

As depicted in Fig. 5.1, 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 [34]). 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 [88]), however, they remain costly in practice

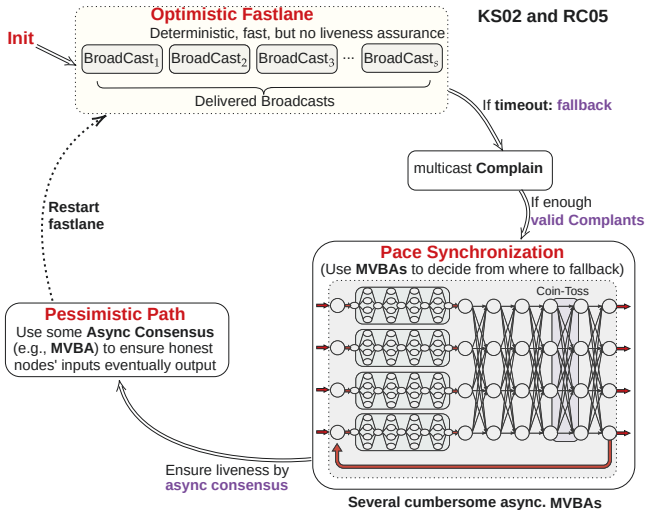


Figure 5.1: Execution flow of KS02 [80] and RC05 [113]. Both rely on cumbersome asynchronous MVBA to do pace-sync.

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 [15]. Later, Aublin et al. [15] 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, [15] used a stronger version of Abstract variant with guaranteed liveness (called Backup). Backup can guarantee all parties to output exact k common transactions [15], 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. [15] also pointed out that Backup (with guaranteed progress) can be obtained from full-fledged BFT protocols. For example, [15] gave exemplary Backup instantiations based on PBFT [42] and Aardvark [48] 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 [73, 74] or have implicit MVBA [66]. That said, though the framework presented in [15] 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 [15]), 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.

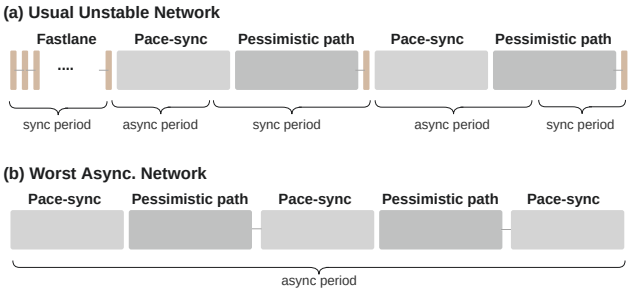


Figure 5.2: Consequence of slow fallback in KS02/RC05 in fluctuating networks. The length of each phase denotes latency.

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.[2] As Fig. 5.2 (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. 5.2 (b), the fallback is

[2]Actual situation might be much worse in RC02 [80] because several more MVBA invocations with much larger inputs are executed in the pace-sync.

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.

5.2 Related work

In the past decades, asynchronous BFT protocols are mostly theoretical results [6, 21–23, 30, 40, 49, 106, 107, 111], until several recent progresses such as HoneyBadgerBFT [92], BEAT [58], Dumbo-BFT protocols [73, 74, 88], VABA [9], DAG-based asynchronous protocols [54, 77], and DispersedLedger [124]. Nevertheless, they still have a latency much larger than that of good-case partially synchronous protocols. Besides the earlier discussed optimistic asynchronous consensus [80, 113] and more general framework [15], Spiegelman recently [120] used VABA [9] 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 [124] and Speeding-Dumbo [73], and can directly employ them to instantiate more efficient pessimistic path.

It is well known that partially synchronous protocols [41, 125] 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 [8, 10, 72, 93, 105, 119] 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 [26–28, 86, 94] 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.

A concurrent work [67] considers adding an asynchronous view-change to a variant of HotStuff. Very recently its extended version [66] was presented with implementations. They focus on a specific construction of asynchronous fallback tailored for HotStuff by opening up a recent MVBA protocol [9], 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, [66] 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.

5.3 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 $\mathsf{block} := \langle epoch, slot, \mathsf{TXs}, \mathsf{Proof}\rangle$, where *epoch* and *slot* are natural numbers, TXs is a sequence of transactions also known as the payload. Throughout the paper, we assume $|\mathsf{TXs}| = B$, 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.

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 $L := |\mathsf{log}|$, we might use hereunder notations. (1) $\mathsf{log}[i]$ denotes the i-th block in log. For example, $\mathsf{log}[1]$ is the

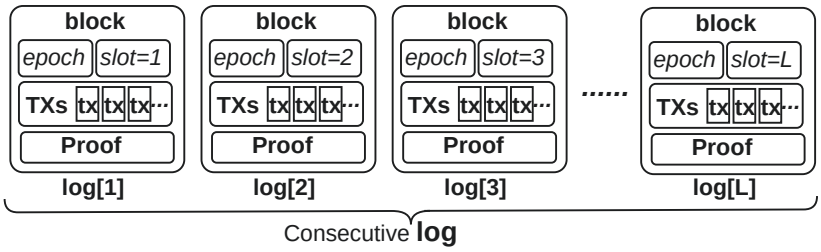


Figure 5.3: Block and output log due to our terminology.

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 $[\mathsf{block}_{x+1}, \ldots, \mathsf{block}_{x+k}]$ as input, |log| would increase by k, and $\mathsf{log}[-1] = \mathsf{block}_{x+k}$, $\mathsf{log}[-2] = \mathsf{block}_{x+k-1}$ 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.

5.3.1 System model

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 [68, 76, 108], and the latter can be feasibly implemented in an asynchronous network as well [7, 55, 56, 64, 79].

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 [58, 73, 74, 92, 124]. 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 [33, 80], 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.

5.3.2 Security goal

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 7. *In atomic broadcast, 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* $\mathsf{log}[i] = \mathsf{log}'[i]$ *for every* i *that* $1 \leq i \leq \min\{|\mathsf{log}|, |\mathsf{log}'|\}$.
- Agreement. *If an honest party adds a* block *to its* log*, all honest parties would eventually add the* block *to their* log*s.*
- Liveness *(adapted from [34]). 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 [34] 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 [21]. Moreover, the protocol must terminate in polynomial number of rounds to restrict the computing steps of adversary in the computationally-secure model [9, 34, 104], otherwise cryptographic primitives are potentially insecure.

5.4 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. 5.5).
- 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 $s+1$.

5.4.1 Overview of the Bolt protocol

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

Definition 8. 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 [12, 41, 59, 72, 125]. 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 [113]. Here we present two more exemplary Bolt constructions.

Comparing with the Abstract *component in [15].* As aforementioned, [15] 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, [15] 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.

Algorithm 10 Bolt from sequential multicasts (Bolt-sCAST) for each party $\mathcal{P}_i$. The external functions are presented in Alg. 12

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 $\ell = (\mathsf{id} \mod n) + 1$)
Initializes $s = 1$, $\sigma_0 = \perp$ and runs the protocol in consecutive slot number s as:

1: **if** $\mathcal{P}_i$ is the leader $\mathcal{P}_\ell$ **then** ▷ Broadcast
2: **if** $s > 1$ **then**
3: 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 $\langle \mathsf{id}, s - 1, \mathcal{H}(\mathsf{TXs}_{s-1})\rangle$
4: compute σ_{s-1}, the full-signature for $\langle \mathsf{id}, s - 1, \mathcal{H}(\mathsf{TXs}_{s-1})\rangle$, by aggregating the $2f + 1$ received valid partial signatures
5: multicast proposal(id, $s, \mathsf{TXs}_s, \sigma_{s-1}$), where $\mathsf{TXs}_s \leftarrow \mathsf{buf}[: B]$
6: **upon** receiving proposal(id, $s, \mathsf{TXs}_s, \sigma_{s-1}$) from $\mathcal{P}_\ell$ **do** ▷ Commit and Vote
7: **if** $s > 1$ **then**
8: proceed only if σ_{s-1} is valid full signature that aggregates $2f + 1$ partial signatures for $\langle \mathsf{id}, s - 1, \mathcal{H}(\mathsf{TXs}_{s-1})\rangle$, otherwise abort
9: output block:=(id, $s - 1, \mathsf{TXs}_{s-1}, \mathsf{Proof}_{s-1}$), where $\mathsf{Proof}_{s-1} := \langle \mathcal{H}(\mathsf{TXs}_{s-1}), \sigma_{s-1}\rangle$
10: send Vote(id, $s, \sigma_{s,i}$) to the leader $\mathcal{P}_\ell$, where $\sigma_{s,i}$ is the partial signature for $\langle \mathsf{id}, s, \mathcal{H}(\mathsf{TXs}_s)\rangle$
11: let $s \leftarrow s + 1$
12: **upon** abandon(id) is invoked **do** ▷ Abandon
13: abort the above execution

5.4.2 Details of the Bolt protocol

Bolt from sequential multicasts. As shown in Alg. 10, 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.

Bolt from sequential reliable broadcast. As shown in Alg. 11, 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

Algorithm 11 Bolt from sequential RBCs (Bolt-sRBC) for each party $\mathcal{P}_i$. The external functions are presented in Alg. 12

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 $\ell = (\mathsf{id} \mod n) + 1$)
Initializes $s = 1$, $\sigma_0 = \perp$ and runs the protocol in consecutive slot number s as:
1: **if** $\mathcal{P}_i$ is the leader $\mathcal{P}_\ell$ **then** ▷ Broadcast
2: activates RBC[⟨id, s⟩] with input $\mathsf{TXs}_s \leftarrow \mathsf{buf}[: B]$;
3: **else**
4: activates RBC[⟨id, s⟩] as non-leader party
5: **upon** RBC[⟨id, s⟩] returns TXs_s **do** ▷ Vote
6: send Vote(id, $s, \sigma_{s,i}$) to all, where $\sigma_{s,i}$ is the partial signature for ⟨id, $s, \mathcal{H}(\mathsf{TXs}_s)$⟩
7: **upon** receiving $2f + 1$ Vote(id, $s, \sigma_{s,i}$) from distinct parties $\mathcal{P}_i$ **do** ▷ Commit
8: **if** $\sigma_{s,i}$ is the valid partial-signature signed by $\mathcal{P}_i$ for ⟨id, $s, \mathcal{H}(\mathsf{TXs}_s)$⟩ **then**
9: let σ_s is the full signature that aggregates the $2f + 1$ partial signatures $\{\sigma_{s,i}\}$
10: let $\mathsf{Proof}_s := \langle \mathcal{H}(\mathsf{TXs}_s), \sigma_s \rangle$
11: output block:=(id, s, TXs_s, Proof_s), and $s \leftarrow s + 1$
12: **upon** abandon(id) is invoked **do** ▷ Abandon
13: abort the above execution

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 [92] can use the technique of verifiable information dispersal [38] 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.

5.4.3 Analyses of the Bolt protocol

. The security analyses of Bolt-sCAST and Bolt-sRBC are simple by nature, and their complexities can be easily counted as well. The detailed analysis is as follows:

Lemma 23. *The algorithm 10 satisfies the* total-order, notarizability, abandonability *and* optimistic liveness *properties of* Bolt *except with negligible probability.*

Proof: Here we prove the four properties one by one:

Algorithm 12 Invocable external functions for Bolt instantiations

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

▷ Leverage the valid Proof_s to extract the s-th block from some received protocol messages
external function Bolt.extract(id, s, Proof_s):
3: **if** Bolt.verify(id, s, Proof_s) = 1 **then**
4: parse Proof_s as $\langle h_s, \sigma_s \rangle$
5: **if** TXs_s was received during executing Bolt s.t. $h_s = \mathcal{H}(\mathsf{TXs}_s)$ **then**
6: return block:=(id, s, TXs_s, Proof_s), where $\mathsf{Proof}_s := \langle h_s, \sigma_s \rangle$
7: **else** return block:=(id, s, $\perp$, $\perp$)

For total-order: First, we prove at same position, for any two honest parties $\mathcal{P}_i$ and $\mathcal{P}_j$ return block_i and block_j, respectively, then $\mathsf{block}_i = \mathsf{block}_j$. It is clear that if the honest party $\mathcal{P}_i$ outputs block_i, then at least $f+1$ honest parties did vote for block_i because $\mathsf{TSIG.Vrfy}_{2f+1}$ passes verification. So did $f+1$ honest parties vote for block_j. That means at least one honest party votes for both blocks, so $\mathsf{block}_i = \mathsf{block}_j$.

For notarizability: Suppose a party $\mathcal{P}_i$ outputs $\mathsf{blocks}[j] := \langle \mathsf{id}, j, \mathsf{TXs}_j, \mathsf{Proof}_j \rangle$, it means at least $f+1$ honest parties vote for $\mathsf{block}[j]$, according to the pseudocode, at least those same $f+1$ honest parties already output $\mathsf{blocks}[j-1]$ and received the TXs_j, hence, those honest parties can further use the valid Proof_j to extract $\mathsf{blocks}[j]$ from the receivied protocol messages.

For abandonability: it is immediate to see from the pseudocode of the abandon interface.

For optimistic liveness: suppose that the optimistic condition is that the leader is honest, then any honest party would output $\mathsf{block}[1]$ in three asynchronous rounds after entering the protocol and would output $\mathsf{log}[j+1]$ within two asynchronous rounds after outputting $\mathsf{log}[j]$ (for all $j \geq 1$). □

Lemma 24. *The algorithm 11 satisfies the* total-order, notarizability, abandonability *and* optimistic liveness *properties of* Bolt *except with negligible probability.*

Proof: It is clear that the *total-order*, *notarizability* and *abandonability* follow immediately from the properties of RBC and the pseudocode of Bolt-sRBC, since the agreement of

RBC guarantees that the output TXs by any parties is the same and a valid proof along with the sequentially executing nature of all RBC instances would ensure total-order and notarizability. *For optimistic liveness*, the optimistic condition remains to be that the leader is honest, and κ is 4 due to the RBC construction in [92] and an extra vote step. □

5.4.4 Overview of the tcv-BA protocol

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 9. 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* $v \in \mathbb{N}$.

Algorithm 13 tcv-BA protocol for each party $\mathcal{P}_i$. Lines different to Alg. 7 in [74] are in orange texts.

For each party $\mathcal{P}_i$, make the following modifications to the ABA code in Alg. 7 of [74] (originally from [99] but with some adaptions to use Ethan MacBrough's suggestion [1] to fix the potential liveness issues of [99]):

Replace line 13-23 of Algorithm 7 in [74] with the next instructions:

1: **upon** $c \leftarrow \mathsf{Coin}_r$.GetCoin() **do**
2: **if** $S_r = \{v\}$ **then**
3: **if** $v\%2 = c\%2$ **then**
4: **if** *decided* = false **then** output v; *decided* = true
5: **else** (i.e, *decided* = true) **then** halt
6: $\mathsf{est}_{r+1} \leftarrow v$
7: **if** $S_r = \{v_1, v_2\}$ **then**
8: **if** $v_1\%2 = c\%2$ **then** $\mathsf{est}_{r+1} \leftarrow v_1$
9: **else** (i.e, $v_2\%2 = c\%2$) **then** $\mathsf{est}_{r+1} \leftarrow v_2$

5.4.5 Details of the tcv-BA protocol

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 ABA construction adapted from [99]. This non-black-box construction basically reuses the

Algorithm 14 tcv-BA protocol built from any ABA "black-box" for each party $\mathcal{P}_i$

Let ABA be any asynchronous binary agreement
1: **upon** receiving input R **do**
2: multicast VALUE(id, R)
3: **upon** receiving VALUE(id, R') from $f+1$ parties containing the same R' **do**
4: **if** VALUE(id, R') has not been sent before **then**
5: multicast VALUE(id, R')
6: **upon** receiving $2f+1$ VALUE(id, v) messages from distinct parties carrying the same v **do**
7: activate ABA[id] with $v\%2$ as input
8: **wait** ABA[id] returns b
9: **if** $v\%2 = b$ **then** return v
10: **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'

protocol pseudocode except several if-else checking (see Alg. 13) and hence has the same performance of this widely adopted ABA protocol.

In addition, tcv-BA can be constructed from any ABA with only 1-2 more "multicast" rounds, cf. Alg 14. This black-box construction provides us a convenient way to inherit any potential improvements of the underlying ABA primitive [3, 50, 56, 126].

5.4.6 Analyses of the tcv-BA protocol

The security analyses of the above two tcv-BA constructions are as follows:

Lemma 25. *The algorithm 13 satisfies the* termination, validity *and* agreement *properties of* tcv-BA *except with negligible probability.*

Proof: For validity, from [99] we know: for any $v \in S_r$, then v was the input of at least one honest party, then in next round $r+1$, every honest party's input est_{r+1} will always from at least one honest party's input of round r by the code. Again, according to the pseudocode, the output is some element in S. Hence, validity is satisfied.

For agreement, suppose that some honest party $\mathcal{P}_i$ is the first party that outputs in some round r and its output is v, for any other honest party, it either outputs the same v in the same round, or has $S_r = \{v, v+1\}$ or $S_r = \{v, v-1\}$. Hence, all honest parties will have the same input $\mathsf{est}_{r+1} = v$ (s.t. $v\%2 = c_r\%2$) in next round $r+1$, then $S_{r+1} = \{v\}$, and thus

$\mathsf{est}_{r+2} = v$ for round $r+2$. Once in some round r', the $v\%2 = c_{r'}\%2$, all honest parties would output the same v. So the agreement is met.

For termination, the analysis is similar to that in [1, 99]. □

Lemma 26. *The algorithm 14 satisfies the* termination*,* validity *and* agreement *properties of* tcv-BA *except with negligible probability.*

Proof: *For termination*: Recall that all honest parties input a value in $\{v, v+1\}$ where $v \in \mathbb{N}$. Without loss of generality, suppose value v was input by at least $f+1$ honest parties. Then, by the code of algorithm, every honest parties can receive $2f+1$ Value(id, v) messages from distinct parties carrying the same v. Hence, all honest parties can activate ABA with some input $v'\%2$. Then, the ABA guarantees that all honest parties return b. Since the validity of ABA guarantees the output of ABA is at least one honest party's input, according to the code, if one honest party input b into ABA, then the party has received at least $2f+1$ Value(id, v') messages from distinct parties carrying the same v' and $v'\%2 = b$, hence, all honest parties can receive $f+1$ Value(id, v') messages from distinct honest parties containing the same v' such that $v'\%2 = b$.

For validity: Since the validity of ABA guarantees the output b of ABA is at least one honest party's input, then according to the code, the party has received at least $2f+1$ Value(id, v') messages from distinct parties carrying the same v', where $v'\%2 = b$, hence, at least one honest party with taking v' as input and multicast Value(id, v').

For agreement: Since the agreement of ABA guarantees all honest parties have the same output b. Hence, all honest parties will output value v', where $v'\%2 = b$. Without loss of generality, suppose honest party $\mathcal{P}_i$ output v and honest party $\mathcal{P}_j$ output $v+2k$ ($k \neq 0$), then following the validity proof, both v and $v+2k$ are honest party's input, then it is a contradiction with the tcv-BA input assumption. □

5.5 Bolt-Dumbo Transformer framework

5.5.1 Overview of the Bolt-Dumbo Transformer framework

At a very high-level, the overview of Bolt-Dumbo Transformer as shown in Fig. 5.4. Let us first briefly walk through how it can overcome kinds of challenges and reduce the complex pace-sync problem to only a variant of asynchronous binary agreement.

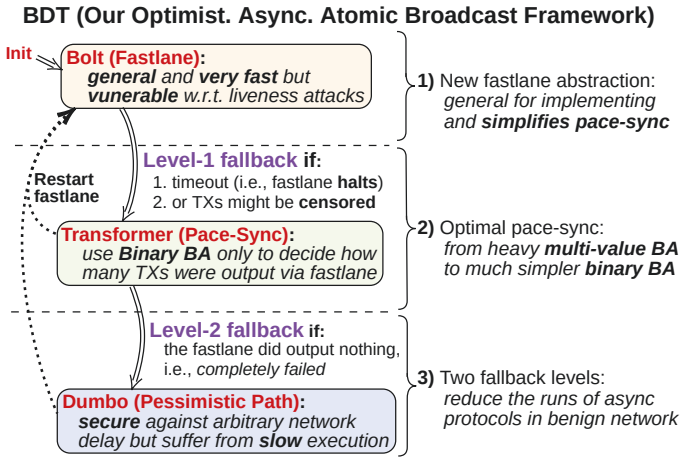


Figure 5.4: The overview of Bolt-Dumbo Transformer.

First ingredient: a new abstraction of the fastlane. In the optimistic case, Bolt (nw-ABC) 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. 5.5. 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

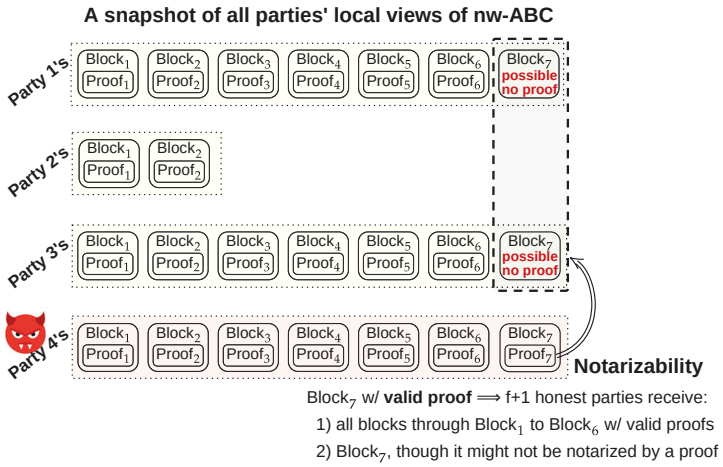


Figure 5.5: Notarizability of fastlane abstraction (nw-ABC).

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 $\{s-1, s\}$. We can handle the problem with the help of tcv-BA, which can be easily implemented from any asynchronous *binary* Byzantine agreement.

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.

Now we present Bolt-Dumbo Transformer (BDT) in details, as Fig. 5.6 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.

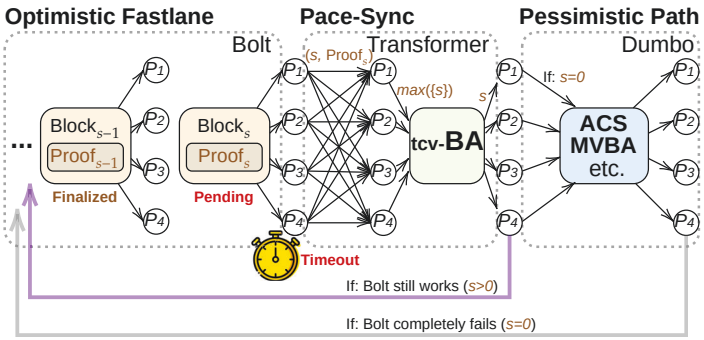


Figure 5.6: The execution flow of Bolt-Dumbo Transformer

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.

5.5.2 Details of the Bolt-Dumbo Transformer framework

BDT is formally illustrated in Alg. 15. It employs a reduction to nw-ABC, tcv-BA, and some asynchronous consensus (e.g., ACS). Informally, it proceeds as follows by successive epochs:

Algorithm 15 The Bolt-Dumbo Transformer (BDT) protocol for each party $\mathcal{P}_i$ in consecutive epoch numbered e (initialized as 1)

/* **Optimistic Path** (also the BDT protocol's main entry) */

Initializes $p_e \leftarrow 0$, $\mathsf{Proof}_e \leftarrow \perp$, $\mathsf{Paces}_e \leftarrow \{\}$, $\mathsf{pending}_e \leftarrow \emptyset$
activate $\mathsf{Bolt}[e]$ instance, and start a timer that expires if not being restarted after τ clock "ticks" (i.e., invoke $timer(\tau).start()$)
upon $\mathsf{Bolt}[e]$ delivers a block **do**
  parse block:=$\langle e, p, \mathsf{TXs}_p, \mathsf{Proof}_p \rangle$, where p is the "slot" number
  ▷ finalized the elder pending block, pending the newly fastlane block
  $\mathsf{log}.append(\mathsf{pending}_e)$, $\mathsf{buf} \leftarrow \mathsf{buf} \setminus \{\text{TXs in } \mathsf{pending}_e\}$, $\mathsf{pending}_e \leftarrow \mathsf{block}$
  $p_e \leftarrow p$, $\mathsf{Proof}_e \leftarrow \mathsf{Proof}_p$, $timer(\tau).restart$ ▷ $\mathsf{Bolt}[e]$ 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 **do**
  invoke $\mathsf{Bolt}[e]$.abandon() and multicast PaceSync$(e, p_e, \mathsf{Proof}_e)$ ▷ the fastlane is stucking or censoring certain txs
upon receiving message PaceSync$(e, p_e^j, \mathsf{Proof}_e^j)$ from $\mathcal{P}_j$ for the first time **do**
  if $\mathsf{Bolt.verify}(e, p_e^j, \mathsf{Proof}_e^j) = 1$ **then** $\mathsf{Paces}_e \leftarrow \mathsf{Paces}_e \cup p_e^j$
  ▷ enough parties have already quitted the fastlane
  if $|\mathsf{Paces}_e| = n - f$ **then**
    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 $\mathsf{maxPace}_e \leftarrow \max(\mathsf{Paces}_e)$ and then $\mathsf{syncPace}_e \leftarrow \mathsf{tcv\text{-}BA}[e](\mathsf{maxPace}_e)$
  if $\mathsf{syncPace}_e > 0$ **then**
    send PaceSync$(e, \mathsf{syncPace}_e, \mathsf{Proof})$ to all if $\mathsf{syncPace}_e \in \mathsf{Paces}_e$, where $\mathsf{Bolt.verify}(e, \mathsf{syncPace}_e, \mathsf{Proof}) = 1$
    if $\mathsf{syncPace}_e = p_e$: $\mathsf{log}.append(\mathsf{pending}_e)$ and $\mathsf{buf} = \mathsf{buf} \setminus \{\text{TXs in } \mathsf{pending}_e\}$
    if $\mathsf{syncPace}_e = p_e + 1$ **then**
      ▷ try to extract the missing block
      wait for a valid PaceSync$(e, \mathsf{syncPace}_e, \mathsf{Proof})$, then $\mathsf{block}' \leftarrow \mathsf{Bolt.extract}(e, \mathsf{syncPace}_e, \mathsf{Proof})$
      ▷ failed to extract, have to rely on other parties to fetch, cf. Alg. 16
      if block' is in form of $(e, \mathsf{syncPace}_e, \perp, \perp)$, then: $\mathsf{block}' \leftarrow \mathsf{CallHelp}(e, p_e, 1)$
      $\mathsf{log}.append(\mathsf{pending}_e).append(\mathsf{block}')$ and $\mathsf{buf} = \mathsf{buf} \setminus \{\text{TXs in } \mathsf{pending}_e \text{ and } \mathsf{block}'\}$
    ▷ contact other parties to fetch missing fastlane blocks, cf. Fig. 16
    if $\mathsf{syncPace}_e > p_e + 1$: $\mathsf{blocks} \leftarrow \mathsf{CallHelp}(e, p_e, \mathsf{syncPace}_e - p_e)$ **then**
      $\mathsf{log}.append(\mathsf{pending}_e).append(\mathsf{blocks})$ and $\mathsf{buf} = \mathsf{buf} \setminus \{\text{TXs in } \mathsf{pending}_e \text{ and blocks}\}$
    continue **Optimistic Path** with $e \leftarrow e + 1$
  if $\mathsf{syncPace}_e = 0$ **then**
    invoke **Pessimistic(e)** and wait for its return, then continue **Optimistic Path** with $e \leftarrow e + 1$

/* **Pessimistic Path** */

internal function Pessimistic(e): ▷ **function** shares all internal states of the BDT protocol
  $\mathsf{txs}_i \leftarrow$ randomly select $\lfloor B/n \rfloor$-sized transactions from the first B-sized transactions at the top of buf
  $x_i \leftarrow \mathsf{TPKE.Enc}(epk, \mathsf{txs}_i)$, namely, encrypt txs_i to obtain x_i
  $\{x_j\}_{j \in S} \leftarrow \mathsf{ACS}[e](x_i)$, 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 $\mathsf{TXs} = \bigcup_{j \in S} \mathsf{txs}_j \leftarrow \{\mathsf{TPKE.Dec}(epk, x_j)\}_{j \in S}$
  let $\mathsf{block} := \langle e, 1, \mathsf{TXs}, \perp \rangle$, then $\mathsf{log}.append(\mathsf{block})$ and $\mathsf{buf} = \mathsf{buf} \setminus \{\text{TXs in block}\}$

This pessimistic path might be replaced by many asynchronous consensus protocols, e.g., ACS and MVBA. Throughout the paper, the pessimistic path is instantiated by Dumbo, i.e., using a reduction to ACS (Honeybadger [92]) and an ACS implementation from Dumbo [74]. Essentially, the underlying ACS [74] can be replaced by any improved implementation.

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. Alg. 16).

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 Alg. 16 illustrates, CallHelp messages specify which blocks to

Algorithm 16 Help and CallHelp. Help is a daemon process having access to the output log, and CallHelp is a function to call Help

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

/* **The CallHelp function** */

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

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 [38, 92], 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 [124] 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.

5.5.3 Analyses of the Bolt-Dumbo Transformer framework

First, we brief the security intuitions of BDT in the following, and then we provide deailed proofs.

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 $s - 1$ 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.

We first prove the total-order and agreement, assuming the underlying Bolt, tcv-BA and ACS are secure.

Claim 1. *If an honest party activates* tcv-BA[e]*, then at least* $n - 2f$ *honest parties have already invoked* abandon(e)*, and from now on: suppose these same parties invoke* abandon(e) *before they output* block:=$\langle e, R, \cdot, \cdot \rangle$*, then any party (including the faulty ones) cannot receive (or forge) a valid* block:=$\langle e, R+1, \cdot, \cdot \rangle$*, and all honest parties would activate* tcv-BA[e].

Proof: When an honest party $\mathcal{P}_i$ activates tcv-BA[e], it must have received $n - f$ valid PaceSync messages from distinct parties, so there would be at least $n - 2f$ honest parties multicast PaceSync messages. By the pseudocode, it also means that at least $n - 2f$ honest parties have invoked abandon(e). Note that $n - 2f \geq f + 1$ and these same parties invoke abandon(e) before they output block:=$\langle e, R, \cdot, \cdot \rangle$, so no party would deliver any valid block:=$\langle e, R + 1, \cdot, \cdot \rangle$ in this epoch's Bolt phase due to the *abandonability* property of Bolt. It is also implies that any parties cannot from the Bolt receive valid block:=$\langle e, R + 1, \cdot, \cdot \rangle$, then all honest parties will eventually be interrupted by the "timeout" mechanism after τ asynchronous rounds and then multicast PaceSync messages. This ensures that all honest parties finally receive $n - f$ valid PaceSync messages from distinct parties, causing all honest parties to activate tcv-BA[e]. □

Claim 2. *Suppose that some party receives a valid* block:=$\langle e, R, \cdot, \cdot \rangle$ *from* Bolt[e] *when an honest party invokes* tcv-BA[e] *s.t. this* block *is the one with largest slot number among all parties' valid* pending *blocks (which means the union of the honest parties' actual* pending *blocks and the malicious parties' arbitrary valid* Bolt[e] *block), then all honest parties'* maxPace$_e$ *must be either* R *or* $R - 1$.

Proof: Following Claim 1, once an honest party invokes tcv-BA[e], the Bolt[e] block with the largest slot number R_{max} would not change anymore. Let us call this already

fixed Bolt[e] block with highest slot number as block_{max}. Since there is someone that receives block_{max}:=$\langle e, R, \cdot, \cdot\rangle$, at least $f+1$ honest parties (e.g., denoted by Q) have already received the block $\langle e, R-1, \cdot, \cdot\rangle$, which is because of the *notarizability* property of Bolt. So these honest parties would broadcast a valid PaceSync$(e, R-1, \cdot)$ message or a valid PaceSync$(e, R, \cdot)$ message. According to the pseudocode in Algorithm 15, $\mathsf{maxPace}_e$ is the maximum number in the set of Paces_e, where Paces_e contains the slot numbers encapsulated by $n-f$ valid PaceSync messages. Therefore, Paces_e must contain one PaceSync message's slot number from at least $n-2f \geq f+1$ honest parties (e.g., denoted by $\bar{Q}$). All honest parties' local Paces_e set must contain $R-1$ and/or R, because $\bar{Q}$ and Q contain at least one common honest party. Moreover, there is no valid PaceSync message containing any slot larger than R since the proof for that is unforgeable, which means R is the largest possible value in all honest parties' Paces_e. So any honest party's $\mathsf{maxPace}_e$ must be R or $R-1$. □

Claim 3. *No honest party would get a* $\mathsf{syncPace}_e$ *smaller than the slot number of it latest finalized block* log[−1] *(i.e., no block finalized in some honest party's* log *can be revoked).*

Proof: Suppose an honest party invokes tcv-BA[e] and a valid Bolt[e] block $\langle e, R, \cdot, \cdot\rangle$ is the one with largest slot number among the union of the honest parties' actual pending blocks and the malicious parties' arbitrary valid Bolt[e] block. Because of Claim 2, all honest parties will activate tcv-BA[e] with taking either R or $R-1$ as input. According to the *strong validity* of tcv-BA, the output $\mathsf{syncPace}_e$ of tcv-BA[e] must be either R or $R-1$. Then we consider the next two cases:

1. Only malicious parties have this $\langle e, R, \cdot, \cdot\rangle$ block;
2. Some honest party $\mathcal{P}_i$ also has the $\langle e, R, \cdot, \cdot\rangle$ block.

For Case 1) Due to the *notarizability* property of Bolt and this case's baseline, there exist $f+1$ honest parties (denoted by a set Q) have the block $\langle e, R-1, \cdot, \cdot\rangle$ as their local pending. Note that remaining honest parties (denoted by a set $\bar{Q}$) would have local pending block not higher than $R-1$. According to the Algorithm 15, we can state that: (i) if the output is R, then all honest parties will sync their log up to the block $\langle e, R, \cdot, \cdot\rangle$ (which include all

honest parties' local pending); (ii) similarly, if the output is $R-1$, all honest parties will sync up till $\langle e, R-1, \cdot, \cdot \rangle$ (which also include all honest parties' local pending). So in this case, all honest parties (i.e., $\bar{Q} \cup Q$) will not discard their pending block, let alone discard some Bolt that are already finalized to output into log.

For Case 2) Let Q denote the set of honest parties that have the block $\langle e, R, \cdot, \cdot \rangle$ as their local pending. Note the remaining honest parties $\bar{Q}$ would have the pending block not higher than R. In this case, following the Algorithm 15, we can see that: (i) if the output is R, then all honest parties will sync their log up to the block $\langle e, R, \cdot, \cdot \rangle$ (which include all honest parties' local pending); (ii) similarly, if the output is $R-1$, all honest parties will sync up to $\langle e, R-1, \cdot, \cdot \rangle$ (which include $\bar{Q}$ parties' local pending and Q parties' finalized output log). So in this case, all honest parties (i.e., $\bar{Q} \cup Q$) will not discard any block in their finalized log. □

Claim 4. *If* tcv-BA[e] *returns* syncPace$_e$*, then at least* $f+1$ *honest parties can append* block*s with slot numbers from* 1 *to* syncPace$_e$ *that all received from* Bolt[e] *into the* log *without invoking* CallHelp *function.*

Proof: Suppose tcv-BA[e] returns syncPace$_e$, then from the *strong validity* of tcv-BA, at least one honest party inputs the number syncPace$_e$. The same honest party must receive a valid message PaceSync(e, syncPace$_e$, Proof), which means there must exists a valid Bolt block $\langle e, \text{syncPace}_e, \cdot, \text{Proof} \rangle$. By the code, the honest party will multicst PaceSync(e, syncPace$_e$, Proof) if tcv-BA[e] returns syncPace$_e$, then all honest parties can get the Proof. Following the *notarizability* and *total-order* properties of Bolt, at least $f+1$ honest parties can append blocks from $\langle e, 1, \cdot, \cdot \rangle$ to $\langle e, \text{syncPace}_e, \cdot, \cdot \rangle$ into the log without invoking CallHelp function. □

Claim 5. *If an honest party invokes* CallHelp *function to retrieve a block* log[i]*, it eventually can get it; if another honest party retrieves a block* log[i]$'$ *at the same log position i from the* CallHelp *function, then* log[i] = log[i]$'$.

Proof: Due to Claim 4 and *total-order* properties of Bolt, any block log[i] that an honest party is retrieving through CallHelp function shall have been in the output log of at least

$f + 1$ honest parties, so it eventually can get $f + 1$ correct HELP messages with the same Merkle tree root h from distinct parties, then it can interpolate the $f+1$ leaves to reconstruct $\mathsf{log}[i]$ which is same to other honest parties' local $\mathsf{log}[i]$. We can argue the agreement by contradiction, in case the interpolation of honest party fails or it recovers a block $\mathsf{log}'[i]$ different from the the honest party's local $\mathsf{log}[i]$, that means the Merkle tree with root h commits some leaves that are not coded fragments of $\mathsf{log}[i]$; nevertheless, there is at least one honest party encode $\mathsf{log}[i]$ and commits the block's erasure code to have a Merkle tree root h; so the adversary indeed breaks the collision resistance of Merkle tree, implying the break of the collision-resistance of hash function, which is computationally infeasible. So all honest parties that attempt to retrieve a missing block $\mathsf{log}[i]$ must fetch the block consistent to other honest parties'. □

Lemma 27. *If all honest parties enter the* Bolt *phase with the same* log*, then they will always finish the* Transformer *phase with still having the same* log′.

Proof: If all honest parties enter the epoch with the same log, it is easy to see that they all will eventually interrupt to abandon the Bolt phase. Due to Claim 1, all honest parties would activate tcv-BA[e]. Following the *agreement and termination* of tcv-BA[e], all parties would finish tcv-BA[e] to get a common $\mathsf{syncPace}_e$, and then by the pseudocode, all honest parties will sync up to the same log, and the last block of log with slot number $\mathsf{syncPace}_e$ (due to *total-order* properties of Bolt, Claim 4 and Claim 5), hence, all parties will finish the Transformer phase with the same output log. □

Lemma 28. *For any two honest parties before finishing the Transformer phase, then there exists one party, such that its* log *is a prefix of (or equal to) the other's.*

Proof: The blocks outputted before the completion of the Transformer phase were originally generated from the Bolt phase, then the Lemma holds immediately by following the *total-order* property of Bolt and Claim 3. □

Lemma 29. *If any honest party enters the* Pessimistic *phase, then all honest parties will enters the phase and always leave the phase with having the same* log.

Proof: If any honest party enter the Pessimistic phase, all honest parties would enter this phase, which is due to Claim 1, the *agreement and termination* property of tcv-BA and $\mathsf{syncPace}_e = 0$. Let us assume that all honest parties enter the Pessimistic phase with the same log, it would be trivial too see the statement for the *agreement and termination* properties of ACS and the *correct and robustness* properties of threshold public key encryption. Then considering Lemma 27 and the simple fact that all honest parties activate with the common empty log, we can inductively reason that all honest parties must enter any Pessimistic phase with the same log. So the Lemma holds. □

Lemma 30. *For any two honest parties in the same epoch, there exists one party, such that its* log *is a prefix of (or equal to) the other's.*

Proof: If two honest parties do not enter the Pessimistic phase during the epoch, both of them only participate in Bolt or Transformer, so this Lemma holds immediately by following Lemma 28. For two honest parties that one enters the Pessimistic phase and one does not, this Lemma holds because the latter one's log is either a prefix of the former one's or equal to the former one's due to Lemma 27 and 28. For the remaining case that both honest parties enter the Pessimistic phase, they must initially have exactly same log (due to Lemma 27). Moreover, in the phase, all honest parties would execute the ACS instances in a sequential manner (e.g., there is only one ACS instance in our exemplary pseudocode), so every honest party would output in one ACS instance only if it has already outputted in all earlier ACS instances. Besides, any two honest party would output the same transaction batch in every ACS instance for the agreement property of ACS. So this Lemma also holds for any two honest parties that are staying in the same epoch. □

Theorem 7. *The* Dumbo-MVBA *protocol satisfies the* agreement *and* total order *properties.*

Proof: The total order be induced by Lemma 30 along with the fact the protocol is executed epoch by epoch. The agreement follows immediately from Lemma 27 and 29 along with the fact that all honest parties initialize with the same empty log to enter the first epoch's Bolt phase. □

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.

Then we prove the liveness property of BDT.

Lemma 31. *If all honest parties enter the* Bolt *phase, once the liveness failed, then they will leave the phase in at most polynomial number of asynchronous rounds and also all enter the* Transformer *phase.*

Proof: The liveness failed in the Bolt phase, it could be either (1). no progress within τ time or (2). some oldest transactions is not output within T time. For (1), at worst case, all honest parties' timeout will interrupt, causing them to abandon the Bolt phase in at most $O(\tau)$ asynchronous rounds. For (2), it will take at most $O(T)$ asynchronous rounds to leave the Bolt phase if there is a suspiciously censored tx in all honest parties' buffers due to some timeout parameter T. Hence, once the liveness failed, all honest parties will leave the phase in at most $O(\tau + T)$ asynchronous rounds. After that, the broadcast of PaceSync message will take one more asynchronous round. After that, all honest parties would receive enough PaceSync messages to enter the Transformer phase, which costs at most $O(\tau + T + 1)$ asynchronous rounds. □

Lemma 32. *If all honest parties enter the* Transformer *phase, they all leave the phase in expected constant asynchronous rounds and then either enter the* Pessimistic *phase or enter the next epoch's* Bolt *phase.*

Proof: If all honest parties enter the Transformer phase, it is trivial to see the Lemma since the underlying tcv-BA terminates in on-average constant asynchronous rounds. If the output of tcv-BA equal 0, then enter the Pessimistic phase, otherwise, enter the next epoch's Bolt phase. □

Lemma 33. *If all honest parties enter the* Pessimistic *phase, all honest parties will leave this* Pessimistic *phase in on-average constant asynchronous rounds with outputting some blocks containing on-average $O(B)$-sized transactions.*

Proof: Similar to [92]'s analysis, Pessimistic phase at least outputs $O(B)$-sized transactions (without worrying that the adversary can learn any bit about the transactions to be outputted before they are actually finalized as output) for each execution. Here we remark that the original analysis in [92] only requires IND-CPA security of threshold public key encryption might be not enough, since we need to simulate that the adversary can query decryption oracle by inserting her ciphertext into the ACS output. Moreover, the underlying Dumbo ACS construction [74] ensures all parties to leave the phase in on-average constant asynchronous rounds. □

Theorem 8. *The* Dumbo-MVBA *protocol satisfies the* liveness *property.*

Proof: Due to Lemma 31 and 32, the adversary would not be able to stuck the honest parties during the Bolt and Transformer phases. Even if in the worst cases, the two phases do not deliver any useful output and the adversary intends to prevent the parties from running the Pessimistic phase (thus not eliminating any transactions from the honest parties' input buffer), we still have a timeout mechanism against censorship, which can ensure to execute the Pessimistic phase for every $O(T)$ asynchronous rounds if there is a suspiciously censored tx in all honest parties' buffers due to some timeout parameter T. Recall Lemma 33, for each tx in all honest parties' buffers, it would take $O(XT/B)$ asynchronous rounds at worst (i.e., we always rely on the timeout T to invoke the Pessimistic phase) to make tx

be one of the top B transactions in all parties' buffers, where X is the bound of buffer size (e.g., an unfixed polynomial in λ). After that, any luckily finalized optimistic phase block would output tx (in few more δ), or still relying on the timeout to invoke the Pessimistic phase, causing the worst case latency $O((X/B + \lambda)\delta T)$, which is a function in the actual network delay δ factored by some (unfixed) polynomial of security parameters. □

5.6 Implementation and Evaluation

To demonstrate the practical performance of BDT, we implement the framework using Dumbo-BFT [74] 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.

5.6.1 Implementation setup

We program the proof-of-concept implementations of BDT, Dumbo and 2-chained 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 [29] (implemented over MNT224 curve). For quorum proofs, we concatenate ECDSA signatures (implemented over secp256k1 curve). For threshold public key encryption, the hybrid encryption approach implemented in HoneyBadger BFT is used [92]. 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 proof-of-concept codebase 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.

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.

5.6.2 Evaluations in the WAN setting

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. 5.7). 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

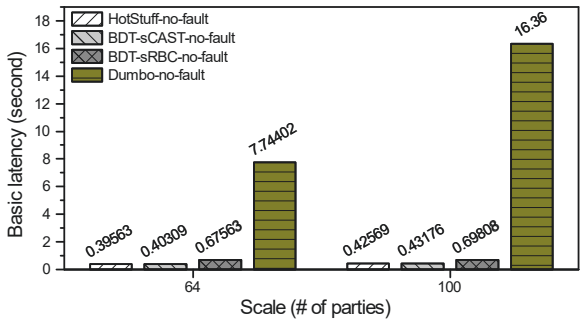


Figure 5.7: Basic latency in experiments over WAN for two-chain HotStuff, BDT-sCAST, BDT-sRBC and Dumbo.

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 [3], 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. 5.8, and gives us an insight how well BDT, HotStuff and Dumbo can handle transaction burst. [4]

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.

[3]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.

[4]Note that we didn't implement an additional layer of mempool as in [63] and [54], and we can expect much higher throughout if we adopt their mempool techniques.

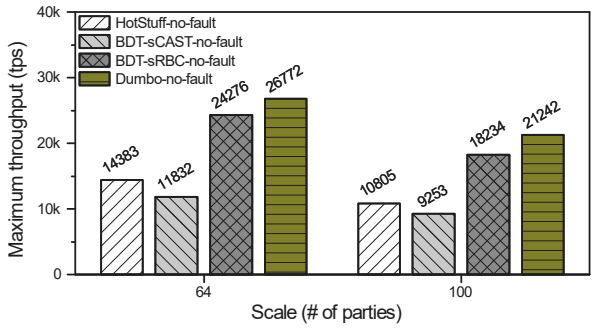


Figure 5.8: Peak throughput in experiments over WAN for two-chain HotStuff, BDT-sCAST, BDT-sRBC and Dumbo.

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. 5.9 and 5.10.

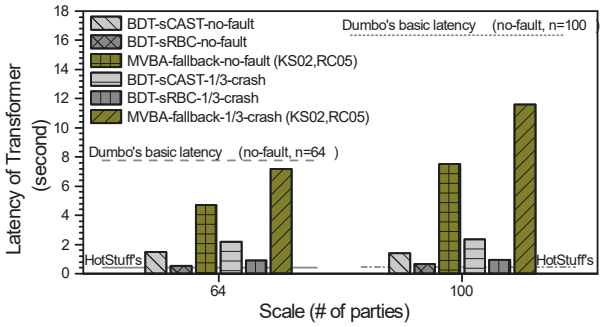


Figure 5.9: Latency of Transformer for pace-sync in BDT-sCAST and BDT-sRBC (when no fault and 1/3 crash, respectively). MVBA fallback in RC05 is also tested as a reference point.

As shown in Fig. 5.9, we measure the execution time of Transformer in various settings by taking combinations of the following setups: (i) BDT-sCAST or BDT-sRBC; (ii) 1/3 crashes on or off; (iii) 64 EC2 instances or 100 EC2 instances. Moreover, in order to comprehensively compare Transformer with the prior art [15, 80, 113], we also measure the latency of MVBA pace-sync (which instantiates the Backup/Abstract primitive in [15] to combine the fastlane and Dumbo [5]) as a basic reference point, cf. Section 2 for the

[5]Following [15] 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 [34, 113] 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

idea of using Backup/Abstract for asynchronous fallback [15]. 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.

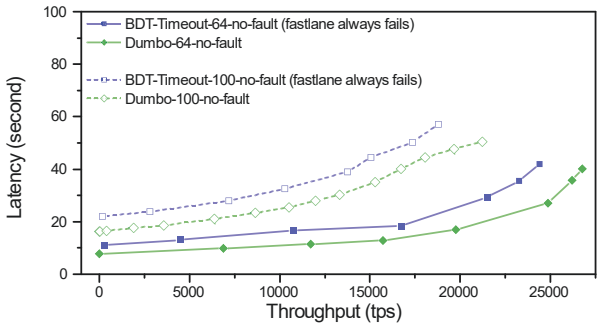


Figure 5.10: Latency v.s. throughput for experiments of BDT with idling fastlane (i.e., fastlane just timeouts after 2.5 sec).

As illustrated in Fig. 5.10, 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).

Latency-throughput trade-off. Figure 5.11 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

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.

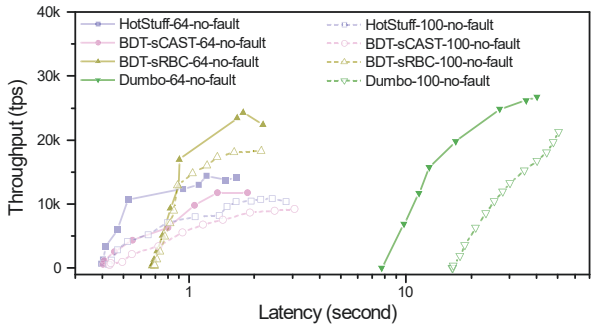


Figure 5.11: Throughput v.s. latency for experiments over WAN when $n = 64$ and 100, respectively (in case of periodically running pace-sync in BDT per only 50 fastlane blocks).

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.

For sake of completeness, we also measure (i) latency and throughput on varying batch sizes and (ii) the latency-throughput trade-off (with $n/3$ faults) in the WAN experiment setting, and plot the results as follows.

Varying batch sizes. For understanding to what an extent the batch size matters, we report how throughput and latency depend on varying batch sizes in Figure 5.12 and 5.13, respectively.

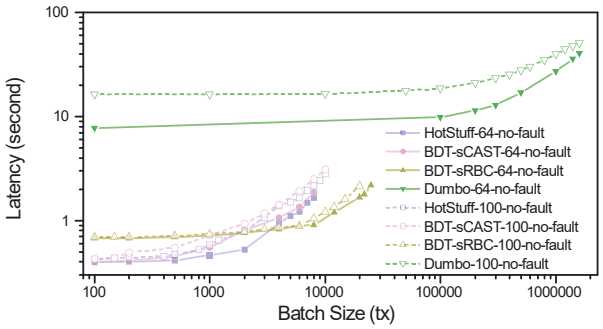


Figure 5.12: Latency v.s. batch size for experiments over wide-area network when $n = 64$ and $n = 100$, respectively.

Figure 5.12 illustrates how latency increases with larger batch size in BDT-sCAST, BDT-sRBC, HotStuff and Dumbo when $n = 64$ and $n = 100$, respectively. It clearly states that: Dumbo always takes a latency much larger than BDT-sCAST, BDT-sRBC and HotStuff; for BDT-sRBC, its latency increases much slower than BDT-sCAST and HotStuff, in particular when $B = 10000$, the latency of BDT-sRBC is around one second only, while these of BDT-sCAST and HotStuff have been more than 2 seconds. The slow increasing of BDT-sRBC's latency is mainly because its better balanced network load pattern.

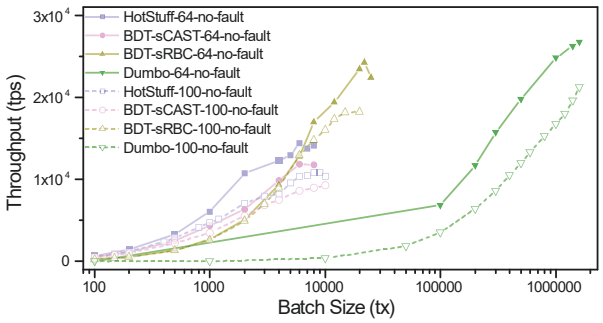


Figure 5.13: Throughput v.s. batch size for experiments over wide-area network when $n = 64$ and $n = 100$, respectively.

Figure 5.13 illustrates how throughput increases with larger batch size in BDT-sCAST, BDT-sRBC, HotStuff and Dumbo when $n = 64$ and $n = 100$, respectively. Dumbo really needs very large batch size to have acceptable throughput; BDT-sCAST and HotStuff have a similar trend that the throughput would stop to increase soon after the batch sizes become larger (e.g., 10000 transactions per block); in contrast, the throughput of BDT-sRBC is increasing faster than those of BDT-sCAST and HotStuff, because larger batch sizes in BDT-sRBC would not place much worse bandwidth load on the leader, and thus can raise more significant increment in the throughput.

Latency-throughput trade-off (1/3 crashes). We also report the latency-throughput trade-off in the presence of $1/3$ crashes. The crashes not only lag the execution of all protocols, but also mimic that a portion of Bolt instances are under denial-of-services. We might fix the batch size of the pessimistic path in BDT-sCAST and BDT-sRBC as 10^6 transactions in these tests, because this batch size parameter brings reasonable throughput-latency trade-off in Dumbo. Shown in Figure 5.14, both BDT-sCAST and

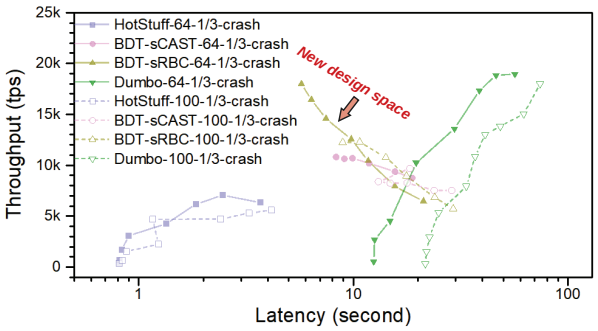


Figure 5.14: Throughput v.s. latency for experiments over wide-area network when $n = 64$ and $n = 100$, respectively (in case of 1/3 crash fault). We fix the fallback batch size of BDT instances to 10^6 transactions in all tests.

BDT-sRBC have some design spaces that show a latency better than Dumbo's and presents a throughput always better than HotStuff's, despite that on average $1/3$ instances of Bolt are unluckily stuck to wait for 2.5 sec to timeout without returning any optimistic output. That means our practical BDT framework does create new design space to harvest the best of both paths, resulting in that we can achieve reasonable throughput and latency simultaneously in fluctuating deployment environments. [6]

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:

1. As fast as 2-chain HotStuff in the best case (i.e., synchronous network without faulty parties); [7]

2. As robust as the underlying asynchronous pessimistic path in the worst case (i.e., the fastlane always completely fails).

[6] We would like to note that here we did a very pessimistic evaluation for BDT while optimistic evaluation for HotStuff in the sense that we manually trigger Transformer by manually muting a leader for 2.5s once in 50 blocks, while for HotStuff we did a stable leader version (with honest leader). In reality, the performance curves for BDT might be a bit more to the left, while HotStuff will surely be more to the right/bottom.

[7] 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.

5.6.3 More evaluations in the controlled dynamic network setting

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 [15] 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.

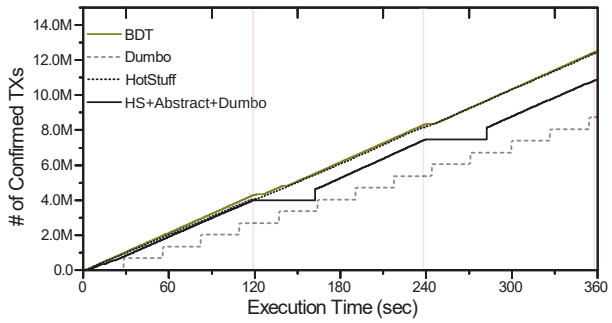


Figure 5.15: Sample executions of BDT, 2-chain HotStuff, Dumbo, and the composition of HotStuff+Abstract+Dumbo for n=64, when facing a few 2-second bad periods. The red region represents the 2-second period of bad network.

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. 5.15. 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.

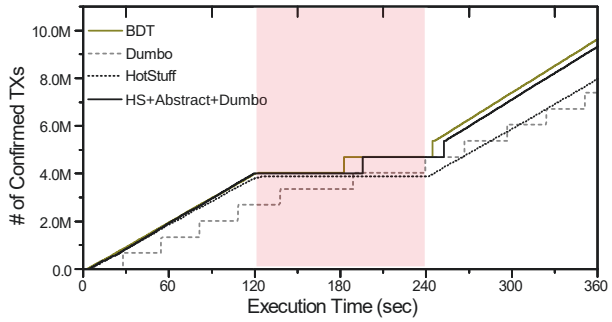


Figure 5.16: Sample executions of BDT, 2-chain HotStuff, Dumbo, and the composition of HotStuff+Abstract+Dumbo for n=64, when suffering from 120-second bad network. The red region represents the 120-second period of bad network.

Intermittent network with long bad time. We then evaluate the effect of long-lasting bad network condition. We visualize such sample executions in Fig. 5.16. 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,

- 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.

5.7 Discussions

5.7.1 Complexity and Numerical Analyses

This section discusses the critical complexity metrics of the BDT framework and those of its major modules. The complexities can be analyzed by counting these of each underlying module. Overall, BDT would cost (expected) $O(n)$ communicated bits per output transaction, and the latency of each output block is of expected constant rounds. These complexities hold in all cases (no matter the network is synchronous or asynchronous). Besides, we then assign each module a running time cost according to our real-world experimental data, thus enabling more precise numerical analysis to estimate the expected latency of BDT in various "simulated" unstable deployment environments.

Complexity analysis. Here we analyze BDT regarding its complexities. Recall that we assume the batch size B sufficiently large, e.g., $\Omega(\lambda n^2 \log n)$, throughout the paper.

Complexities of the fastlane (also of the optismtic cases). For the optimistic fastlane, we have two instantiations, namely Bolt-sCAST and Bolt-sRBC. As shown in Table 5.1, Bolt-sCAST is with linear $O(n)$ per-block message complexity, and the leader's per-block bandwidth usage $O(nB)$ is also linear in n; Bolt-sRBC is with quadratic per-block message complexity as $O(n^2)$, while the per-block bandwidth usage of every party is not larger than the batch size $O(B)$. We can also consider their latency in term of "rounds" to generate a block, i.e., the time elapsed between when a block's transaction is first multicasted and when the honest parties output this block with valid proof. The latency of generating two successive blocks can also be considered to reflect the confirmation latency of BDT's fastlane. Though both "fastlane" instantiations will cost $O(1)$ rounds to generate fastlane blocks, Bolt-sCAST has slightly less concrete rounds: Bolt-sCAST can use at most 3 rounds to generate one (pending) block and can use 5 rounds to output two successive blocks (thus the former block can be finalized in BDT framework); Bolt-sRBC would cost 4 rounds to generate one (pending) block and use 8 rounds to output two successive blocks.

Note that in the *optimistic case* when (i) the fastlane leaders are always honest and (ii) the network condition is benign such that the fastlanes never timeout, the Pessimistic phase is not executed, so the fastlane cost shown in Table 5.1 would also reflect the amortized complexities of the overall BDT protocol (in case that the epoch size $Esize$ is large enough, e.g., n).

Table 5.1: Per-block performance of different Bolt instantiations (which is also per-block cost of BDT in the good cases)

	Msg.	Comm.	Per-block latency	Two blocks latency	Bandwidth Cost	
					Leader	Others
Bolt-sCAST	$O(n)$	$O(nB)$	3 rounds	5 rounds	$O(nB)$	$O(B)$
Bolt-sRBC	$O(n^2)$	$O(nB)$	4 rounds	8 rounds	$O(B)$	$O(B)$

Worst-case complexities disregarding the adversary. In the optimistic fastlane, there is a worst-case overhead of using $O(\tau)$ asynchronous rounds to leave the tentatively optimistic execution without outputting any valid blocks. After the stop of fastlane, all parties enter the Transformer phase, and would participate in tcv-BA, in which the expected message complexity is $O(n^2)$, the expected communication complexity is $O(\lambda n^2)$, and the expected bandwidth cost of each parties is $O(\lambda n)$. Besides, if the output value of tcv-BA is large than zero, then the CallHelp subroutine could probably be invoked, this process will incur $O(n^2)$ overall message complexity and $O(nB)$ per-block communication complexity and causes each party to spend $O(B)$ bandwidth to fetch each block on average. In the worst case, Dumbo is executed after Transformer, which on average costs overall $O(n^3)$ messages,[8] overall $O(nB)$ communicated bits, and $O(B)$ bandwidth per party for each block if batch size B is sufficiently large. The latency of generating a block in the pessimistic path is of $O(1)$ rounds on average.

To summarize these, we can have the worst-case performance illustrated in Table 5.2. Note that the latency of generating a block shall consider the following possible worst case: the fastlane times out to run pace-sync, but pace-sync finalizes no fastlane block, and all parties

[8]Note that if instantiating the pessimistic path by more recent asynchronous BFT consensus protocols (e.g., Speeding Dumbo) instead of Dumbo-BFT, the $O(n^3)$ per-block messages can be reduced to $O(n^2)$.

Table 5.2: Per-block performance of BDT in the worst cases

	Msg.	Comm.	Block latency (rounds)	Bandwidth Cost	
				Leader	Others
BDT-sCAST	$O(n^3)$	$O(nB)$	$3+1+T_{\text{tcv-BA}}+T_{\text{Dumbo}}$	$O(nB)$	$O(B)$
BDT-sRBC	$O(n^3)$	$O(nB)$	$4+1+T_{\text{tcv-BA}}+T_{\text{Dumbo}}$	$O(B)$	$O(B)$

* Note that the worst-case block latency reflects the case of turning off the level-1 fallback.

have to start the pessimistic path to generate a block. Thus, to count the worst-case latency, we need to include: (i) the timeout parameter τ; (ii) the latency of fallback (including 1 round for multicast PaceSync message and the expected latency $T_{\text{tcv-BA}}$ of tcv-BA), and (iii) the expected pessimistic path latency T_{Dumbo}. Here the timeout parameter τ in our system is not necessarily close to the network delay upper bound Δ, and it can represent some adversary-controlling "clock ticks" to approximate the number of asynchronous rounds spent to generate each fastlane block, i.e., $O(\tau) = O(1)$. For example, in BDT-sCAST, τ can approximate 3 rounds because in Bolt-sCAST, the first fastlane block (i.e., the first "heartbeat") needs 3 rounds to deliver and the interval of two successive fastlane blocks (i.e., the interval of two "heartbeats") is 2 rounds; similarly, τ can approximate 4 rounds in BDT-sRBC.

Complexities in comparision to other BFT consensuses. Here we also summarize the communication complexities of BDT and some known BFT protocols in the optimistic case and the worst case, respectively. To quantify the latency of those protocols in unstable network environment, Table 5.3 also lists each protocol's average latency (in "unit" of fastlane's good-case latency).

This metric considers that the fastlane has a probability $\alpha \in [0, 1]$ to output blocks in time, and also has a chance of $\beta = 1 - \alpha$ that falls back and then executes the pessimistic asynchronous protocol. Let C be the latency of using earlier asynchronous protocols [34,92] directly as the pessimistic path and c be that of the state-of-the-art Dumbo BFT [74] and that of using MVBA for synchronization during fallback. Both C and c are represented in the unit of fastlane latency. According to the experimental data [74, 92], C is normally at hundreds and the latter c is typically dozens ([92] runs n instances of ABA, thus rounds

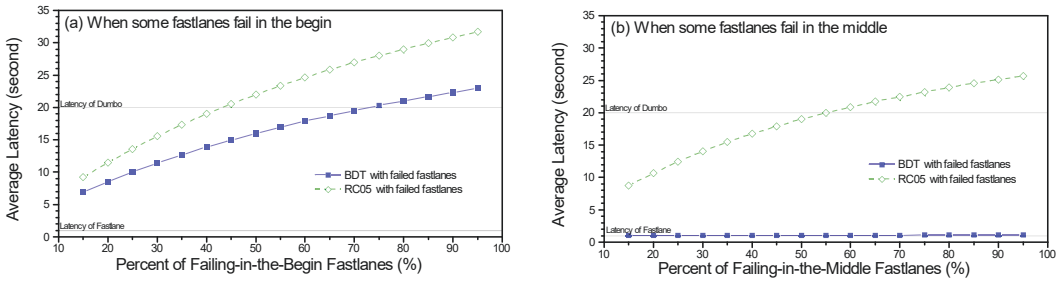
Table 5.3: Complexities of BFT protocols in various settings
(where B is sufficiently large s.t. all λ terms are omitted, and $\alpha + \beta = 1$)

Protocol	Per-block Com. Compl.		Normalized average latency considering fastlane latency as "unit"
	Optim.	Worst	
PBFT [41]	$O(nB)$	∞	$1/\alpha$
HotStuff [125]	$O(nB)$	∞	$1/\alpha$
HBBFT [92]	$O(nB)$	$O(nB)$	C
Dumbo [74]	$O(nB)$	$O(nB)$	c
KS02 [80]	$O(n^2B)$	$O(n^3B)$	$(\alpha + \frac{\beta}{C+kc})^{-1}$ ⋆
RC05 [113]	$O(nB)$	$O(n^3B)$	$(\alpha + \frac{\beta}{C+c})^{-1}$
BDT (ours)	$O(nB)$	$O(nB)$	$(\alpha + \frac{\beta}{c+1})^{-1}$

⋆ There is an integer parameter k in [80] to specify the degree of parallelism for the fastlane, thus probably incurring extra cost of fallback.

depend on number of parties, while [74] reduces it to constant). Our fallback is almost as fast as the fastlane, so its magnitude around one. As such, we can do a simple calculation as shown in Table 5.3 to *roughly estimate* the latency of all those protocols deployed in the realistic fluctuating network.

Numerical analysis on latency in unstable network. To understand the applicability level of BDT framework, we further conduct more precise numerical estimations to visualize the average latency of BDT and prior art (e.g. RC05) in the unstable Internet deployment environment, in particular for some typical scenarios between the best and the worst cases.



(a) When some fastlanes fail in the begin (b) When some fastlanes fail in the middle

Figure 5.17: Numerical analysis to reflect the average latency of BDT and RC05 [113] in fluctuating deployment environment. The analysis methodology is similar to the formulas in Table 5.3 except that here consider more protocol parameters such as batch size, epoch size, timeout, etc.

The real-world experiment data shown in Section 5.6 is considered to specify the cost of each protocol module in the estimations. In particular, we use our experimental results

over the globe when $n = 100$ to specify the parameters used in the numerical estimations regarding both RC05 and BDT: we set the latency of fastlane as 1 second (to reflect the actual latency of Bolt) and set the latency of pessimistic path as 20 seconds (according to the measured latency of Dumbo); the fastlane block and the pessimistic block are set to contain 10^4 and 10^6 transactions, respectively; the MVBA fallback in RC05 is set to use 10 seconds and our Transformer is set to cost 1 second (cf. Figure 5.10); for fair comparison, we let RC05 to use the state-of-the-art Dumbo protocol as its pessimistic path; other protocols parameters (e.g., epoch size and timeout) are also taken into the consideration and are set as same as those in the experiments. Note that a "second" in the simulations is a measurement of virtual time (normalized by the fastlane latency) rather than a second in the real world.

We consider two simulated scenarios. One is illustrated in Figure 5.17 (a), in which there are some portion of fastlane instances that completely fail and output nothing but just timeout and fallback after idling for 2.5 seconds, while the else fastlane instances successfully output all optimistic blocks. In the case, BDT can save up to almost 10 seconds on average latency relative to RC05. This is a result of the much more efficient fallback mechanism; more importantly, the efficient fallback brings much robuster performance against unstable network environment, for example, RC05 starts to perform worse than Dumbo once more than 45% fastlane instances completely fail in the beginning of their executions, while BDT can be faster than Dumbo until more than 75% fastlane instances completely fail. The other case is shown in Figure 5.17 (b), where some fastlane instances stop to progress in the middle of their executions (e.g., stop to progress after 25 optimistic blocks are finalized) and then wait for 2.5 seconds to timeout and fallback. In the case, BDT performs *almost as fast as its underlying fastlane* (i.e., the average delay is really close to 1 second) despite the overheads of timeout and Transformer in this fluctuating network condition; in contrast, RC05 can be an order of magnitude slower than BDT, and it would be even slower than Dumbo if more than 55% fastlane instances fail to progress in the middle of their optimistic executions.

5.7.2 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. We believe these optimistic conditions can ensure the progress of fastlanes.

The main reason is that the fastlane can successfully execute without invoking pace-synchronization, if the following two optimistic conditions hold: (i) the network stays in synchrony, such that the guessed timeout parameter is larger than the "heartbeat" period of the underlying fastlane; (ii) the optimistic liveness condition of fastlane is satisfied, e.g. the leader is honest.

First, it is clear to see: if all honest parties have already enter the same epoch's fastlane at the same time, then the fastlane must successfully progress in the presence of above optimistic conditions. Actually, the above argument still holds, even if the honest parties enter the fastlane with minor difference in time, because one can slightly tune up the guessed timing parameter.

Then, let us briefly argue that when the network is synchronous, BDT can ensure all honest parties to enter the same fastlane within a bounded period. This actually reduces to the next question: when some honest party first outputs and halts in the asynchronous pessimistic path, would all honest parties output soon (if the network is synchronous)? Fortunately, the answer is yes if we check the detailed construction of asynchronous protocols (such as Dumbo). This indicates that the asynchronous pessimistic path itself can work as a clock synchronizer to ensure that all honest parties restart the fastlane nearly at the same time (when network synchrony holds).

5.8 Summary

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. Different from pioneering studies [15, 71, 80, 113] that only demonstrated theoretic feasibility and had questionable practicability because of complex and slow asynchronous pace-synchronization, 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 [15, 71, 80, 113].

- *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 (ABA). In contrast, prior art [80, 113] leveraged cumbersome multi-valued agreement (MVBA) for pace-sync. 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 ABA 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 [80, 113] 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 ABA to further reduce the fallback overhead (e.g., the recent designs from Crain [50], Das et al. [56], Zhang et al. [126] and Abraham et al. [3]). Similarly, though currently we use Dumbo-BFT as the pessimistic path, this can be replaced by more efficient recent designs [73, 124].

CHAPTER 6

SUMMARY OF THE THESIS

6.1 Conclusion

In the WAN setting, asynchronous Byzantine fault tolerant protocols are arguably the most suitable choices for constructing permissioned blockchains with intrusion tolerance and high security assurance. In particular, with the current trend towards a blockchain-based decentralization paradigm, an unprecedented demand calling for practical asynchronous BFT protocols is increasing. Despite the fact that this topic was extensively explored in many earlier studies, most asynchronous attempts concentrated on theoretical feasibility, which is why several attempts performed were only just passable. Aside from that, it is difficult for (partial) synchronous efforts to fulfill the current demand because these protocols are not well suited for the WAN environment. For example, though a recent breakthrough work [92] presented the first practical asynchronous BFT atomic broadcast protocol HBBFT, it still suffers from substantial performance obstacles that restrict its broader application. Actually, designing more performant asynchronous BFT protocols is an interesting and open research problem, and is also one of the hotspots of consensus research.

We presented the novel Dumbo family protocols to resolve many remaining challenges of achieving high-performant asynchronous BFT protocols. We identified the major bottleneck of HBBFT, and then presented the Dumbo protocol to address this pain point via the innovative use of MVBA, which made Dumbo outperform HBBFT. After that, in order to continue improving the efficiency of the Dumbo protocol, we designed a new MVBA protocol Speeding MVBA that achieves fewer concrete rounds, and proposed Speeding Dumbo, a new asynchronous BFT atomic broadcast centering around Speeding MVBA and a cheaper broadcast component.

Besides the brief recall of Dumbo and Speeding Dumbo, this thesis then focuses on the other three recent members of the Dumbo family. First, we discussed MVBA and answered a nearly 20 years open problem affirmatively by presenting Dumbo-MVBA, which has optimal communication complexity when the input size is moderately large. As a result of the high communication complexity of existing MVBA protocols, it was previously thought that employing the MVBA as a building block to instantiate ABC was suboptimal in HBBFT [92]. With the Dumbo-MVBA at hand, we can also demonstrate that MVBA is still the right way to build an ABC with higher asymptotic performance, which completely mitigates the communication blow-up problems in [92].

Second, we discovered that the agreement phase hinders throughput and "wastes" available bandwidth by incurring high latency. Additionally, in order to defend against serious censorship threats, it must also rely on some heavy cryptographic primitives or suffer from reduced efficiency. Focusing on these issues, we presented Dumbo-NG, which resolves the severe tension between throughput and latency and prevents transaction censorship with no extra cost.

Third, we concentrated on real-world network scenarios. In spite of the fact that asynchronous BFT protocols are thought to be robust even when employed in malicious networks, if we always utilize asynchronous BFT protocols in WAN, then the performance will be poor compared with deterministic protocols in the normal case. Because it is possible that the WAN will not always be in adversarial networks, and the deterministic protocols can work very quickly when the network condition is friendly. Therefore, we design a BFT atomic broadcast protocol (BDT) that combines the advantages of the synchronous and asynchronous paradigms, such that it is "as fast as" the current state-of-the-art deterministic BFT consensus on the normal Internet with fluctuations and nearly as fast as the existing performant asynchronous BFT consensus in the worst-case asynchronous network.

All these results from the Dumbo family increase the potential for establishing an actual asynchronous BFT protocol that can eventually be deployed in the real world. Furthermore,

because these works are most frameworks, we can cherry-pick the best instantiations of the related components once these underlying building blocks make progress.

6.2 Future work

Although this thesis presents some of the most recent results on asynchronous BFT protocols, and these results are promising work toward optimal and practical asynchronous BFT protocols, there are still many related difficulties that pose challenges to the performance of asynchronous BFT in practice. The following suggestions are made for potential lines of future research.

The Dumbo family introduced a few different frameworks of asynchronous BFT protocols, and each of them is superior in specific application situations. However, there are still unexplored aspects at the application level. For instance, when there's a requirement to deploy a protocol where MVBA functions as a component, then we need to know in which situations we should instantiate MVBA with the underlying MVBA of Dumbo-MVBA$\star$ rather than Dumbo-MVBA$\star$. In fact, when dealing with a relatively small input size, such as being on the order of $O(\lambda)$, it's possible for the performance of Abraham et al.'s MVBA and Speeding MVBA to outperform that of Dumbo-MVBA$\star$, where Dumbo-MVBA$\star$ serves as an extension of these MVBA protocols. It would be fascinating to see how different scales, input sizes, and underlying cryptography primitives affect the performance of Dumbo-MVBA$\star$ and its underlying MVBA. What's more, it's possible to implement a more efficient asynchronous BFT, one that uses BDT in place of Dumbo-NG's agreement phase. Also, we can select an appropriate MVBA in BDT for certain application scenarios if we have clearly investigated in which situations the underlying MVBA has superior practical performance than Dumbo-MVBA$\star$. This work serves as an immediate follow-up study that will be more effective when instantiating the asynchronous BFT protocol to achieve the best performance when dealing with different situations.

Another fascinating research area involves studying how computation complexity influences protocol latency. In this thesis, we only pay attention to message, communication

and time complexity; the computation complexity is completely ignored. As an important metric, computation complexity has a significant effect on latency. For instance, if we employ (threshold) signatures to lower communication complexity, the trade-off is an increase in computational complexity. This prompts us to evaluate, in practical implementations, whether we can enhance overall performance by (partially) reducing computational expenses.

The purpose of this thesis is to establish optimal and practical asynchronous BFT protocols. The asynchronous BFT community has made fast progress. Except for the methodology introduced in this thesis, another concurrent path to achieve asynchronous BFT atomic broadcast is via the DAG way [19, 53, 54, 69, 70, 77, 98, 103, 109, 116]. However, these works still have some gaps for practical asynchronous BFT. One important reason for this is that the existing asynchronous BFT protocols are not supported for use on a large scale; for instance, the performance drops noticeably when the number of nodes exceeds a few hundred. In reality, the successful applications (such as Bitcoin [100], Ethereum [123], and Aptos [2]) have implemented BFT on a large scale at the Internet level. Due to the fact that the WAN is an asynchronous environment [115], these cryptocurrencies raise security concerns (loss of liveness or loss of safety). To simplify, we can view these successful cryptocurrencies as modern BFT blockchains, and BA-based and DAG-based atomic broadcasts as traditional BFT blockchains. One of the next things that deserve to be explored is the development of a new asynchronous BFT protocol that combines the advantages of the traditional BFT blockchain and the modern BFT blockchain, such that the new protocol with best-of-both-worlds properties in terms of robustness and efficiency.

BIBLIOGRAPHY

[1] Bug in aba protocol's use of common coin #59. https://github.com/amiller/HoneyBadgerBFT/issues/59.

[2] The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. *Aptos white paper*, 2022.

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

[4] Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 317–326, 2019.

[5] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience. In *International Conference on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019.

[6] Ittai Abraham, Danny Dolev, and Joseph Y Halpern. An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-seventh ACM symposium on principles of distributed computing*, pages 405–414, 2008.

[7] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.

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

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

[10] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 331–341, 2021.

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

[12] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of tendermint-core blockchains. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[13] Apache Kafka. http://kafka.apache.org.

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

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

[16] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306, 2013.

[17] Joonsang Baek and Yuliang Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. In *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, volume 3, pages 1491–1495. IEEE, 2003.

[18] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

[19] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 2016.

[20] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 7, 2019.

[21] Michael Ben-Or. Another advantage of free choice (extended abstract) completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

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

[23] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192, 1994.

[24] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[25] Richard E Blahut. *Theory and practice of error control codes*, volume 126. Addison-Wesley Reading, 1983.

[26] Erica Blum, Jonathan Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*, pages 131–150, 2019.

[27] Erica Blum, Jonathan Katz, and Julian Loss. Tardigrade: An atomic broadcast protocol for arbitrary network conditions. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 547–572. Springer, 2021.

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

[29] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

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

[31] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

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

[33] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 2002 ACM SIGSAC Conference on Computer and Communications Security*, pages 88–97, 2002.

[34] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*, pages 524–541. Springer, 2001.

[35] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132, 2000.

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

[37] Christian Cachin and Jonathan A Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings International Conference on Dependable Systems and Networks*, pages 167–176. IEEE, 2002.

[38] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

[39] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[40] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, 1993.

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

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

[43] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[44] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

[45] T-H Hubert Chan, Rafael Pass, and Elaine Shi. Pala: A simple partially synchronous blockchain. *IACR Cryptology ePrint Archive*, 2018.

[46] Brad Chase and Ethan MacBrough. Analysis of the xrp ledger consensus protocol. *arXiv preprint arXiv:1802.07242*, 2018.

[47] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009.

[48] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

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

[50] Tyler Crain. 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.

[51] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: a secure, fair and scalable open blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 466–483. IEEE, 2021.

[52] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. *Atomic broadcast: From simple message diffusion to Byzantine agreement*, volume 118. Elsevier, 1995.

[53] George Danezis and David Hrycyszyn. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620*, 2018.

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

[55] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

[56] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2518–2534. IEEE, 2022.

[57] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

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

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

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

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

[62] Chaya Ganesh and Arpita Patra. Optimal extension protocols for byzantine broadcast and agreement. *Distributed Computing*, 34(1):59–77, 2021.

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

[64] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In *2022 IEEE 42nd International Conference on Distributed Computing Systems*, 2022.

[65] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310. Springer, 2015.

[66] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 296–315. Springer, 2022.

[67] Rati Gelashvili, Lefteris Kokoris-Kogias, Alexander Spiegelman, and Zhuolun Xiang. Be prepared when network goes bad: An asynchronous view-change protocol. *arXiv preprint arXiv:2103.03181*, 2021.

[68] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 295–310. Springer, 1999.

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

[70] Adam Gkagol, Damian Leśniak, Damian Straszak, and Michał Świketek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019.

[71] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376, 2010.

[72] 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 *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

[73] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. In *The 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

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

[75] James Hendricks, Gregory R Ganger, and Michael K Reiter. Verifying distributed erasure-coded data. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 139–146, 2007.

[76] Aniket Kate and Ian Goldberg. Distributed key generation for the internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 119–128. IEEE, 2009.

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

[78] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.

[79] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1751–1767, 2020.

[80] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming*, pages 204–215. Springer, 2005.

[81] Leslie Lamport. The weak byzantine generals problem. *Journal of the ACM (JACM)*, 30(3):668–676, 1983.

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

[83] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science*, 645:1–24, 2016.

[84] Benoît Libert and Moti Yung. Adaptively secure non-interactive threshold cryptosystems. In *International Colloquium on Automata, Languages, and Programming*, pages 588–600. Springer, 2011.

[85] Chao Liu, Sisi Duan, and Haibin Zhang. Epic: efficient asynchronous bft with adaptive security. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 437–451. IEEE, 2020.

[86] Julian Loss and Tal Moran. Combining asynchronous and synchronous byzantine agreement: The best of both worlds. *IACR Cryptology ePrint Archive*, 2018:235, 2018.

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

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

[89] Ethan MacBrough. Cobalt: Bft governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

[90] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1041–1053, 2019.

[91] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[92] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[93] Atsuki Momose, Jason Paul Cruz, and Yuichi Kaji. Hybrid-bft: Optimistically responsive synchronous consensus with optimal latency or resilience. *Cryptology ePrint Archive*, 2020.

[94] Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1686–1699, 2021.

[95] Henrique Moniz, Nuno F Neves, and Miguel Correia. Byzantine fault-tolerant consensus in wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, 12(12):2441–2454, 2012.

[96] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo. Randomization can be a healer: Consensus with dynamic omission failures. In *Distributed Computing: 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings 23*, pages 63–77. Springer, 2009.

[97] Henrique Moniz, Nuno Ferreria Neves, Miguel Correia, and Paulo Verissimo. Ritas: Services for randomized intrusion tolerance. *IEEE transactions on dependable and secure computing*, 8(1):122–136, 2008.

[98] Louise E Moser and Peter M Melliar-Smith. Byzantine-resistant total ordering algorithms. *Information and Computation*, 150(1):75–111, 1999.

[99] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 2–9, 2014.

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

[101] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing*, 2020.

[102] Gil Neiger. Distributed consensus revisited. *Information processing letters*, 49(4):195–201, 1994.

[103] Quan Nguyen, Andre Cronje, Michael Kong, Egor Lysenko, and Alex Guzev. Lachesis: Scalable asynchronous bft on dag streams. *arXiv preprint arXiv:2108.01900*, 2021.

[104] Rafael Pass and Elaine Shi. The sleepy model of consensus. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 380–409. Springer, 2017.

[105] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2018.

[106] Arpita Patra. Error-free multi-valued broadcast and byzantine agreement with optimal communication complexity. In *International Conference On Principles of Distributed Systems*, pages 34–49. Springer, 2011.

[107] Arpita Patra, Ashish Choudhary, and Chandrasekharan Pandu Rangan. Simple and efficient asynchronous byzantine agreement with optimal resilience. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 92–101, 2009.

[108] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 522–526. Springer, 1991.

[109] Serguei Popov. The tangle. *White paper*, 1(3):30, 2018.

[110] Bart Preneel, René Govaerts, and Joos Vandewalle. Cryptographic hash functions: an overview. In *Proceedings of the 6th international computer security and virus conference (ICSVC 1993)*, volume 19, 1993.

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

[112] Michael O Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989.

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

[114] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless bft consensus through metastability. *arXiv preprint arXiv:1906.08936*, 2019.

[115] Muhammad Saad, Afsah Anwar, Srivatsan Ravi, and David Mohaisen. Revisiting nakamoto consensus in asynchronous networks: A comprehensive analysis of bitcoin safety

and chainquality. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 988–1005, 2021.

[116] Maria A Schett and George Danezis. Embedding a deterministic bft protocol in a block dag. *arXiv preprint arXiv:2102.09594*, 2021.

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

[118] Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

[119] Nibesh Shrestha, Ittai Abraham, Ling Ren, and Kartik Nayak. On the optimality of optimistic responsiveness. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 839–857, 2020.

[120] Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. In *35th International Symposium on Distributed Computing (DISC 2021)*, 2021.

[121] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. Mir-bft: High-throughput robust bft for decentralized networks. *arXiv preprint arXiv:1906.05552*, 2019.

[122] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144. IEEE, 2009.

[123] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

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

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

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

[127] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649, 2020.