Efficient and Adaptively Secure Consensus

by

Sravya Yandamuri

Department of Computer Science
Duke University

Defense Date: March 31, 2025

Approved:

Kartik Nayak, Supervisor

Jeffrey Chase

Michael K. Reiter

Jun Yang

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2025

ABSTRACT

Efficient and Adaptively Secure Consensus

by

Sravya Yandamuri

Department of Computer Science
Duke University

Defense Date: March 31, 2025

Approved:

Kartik Nayak, Supervisor

Jeffrey Chase

Michael K. Reiter

Jun Yang

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2025

Abstract

We study the problem of designing consensus protocols with improved latency that are secure when the adversarial conditions change during the course of the protocol. Typically, consensus protocols proceed in a series of communication rounds in which the parties exchange messages with each other or with certain designated parties, and the parties only commit a value at the end of these rounds. Thus, changing adversarial conditions may impede the progress or affect the safety conditions of the underlying consensus protocol. We improve the latency of consensus protocols under two types of adversarial settings.

The first adversary is adaptive, in that it can choose which parties to corrupt during the course of the protocol execution and delay any messages for a finite amount of time. We present a simple framework for consensus that is efficient in the number of rounds till termination against such an adversary in an asynchronous network. In addition, we show new lower and upper bounds on the round complexity of some key primitives used for achieving adaptively secure asynchronous agreement.

The second adversarial setting, known as dynamic availability, allows parties to join or leave the network at any time. In this setting, it is known that the latency should depend on a known pessimistic network delay, and it cannot depend on the speed of the network. In this setting, we present a protocol that achieves latency better than the pessimistic network delay assumption.

Contents

List of Tables

List of Figures

Acknowledgements

To Kartik – thank you for answering my endless questions and for everything you taught me. If not for you, I would not have been introduced fascinating world of consensus.

To my parents, Sudha and Mohan – thank you for asking "Is it done yet?" just the right number of times to keep me going without completely losing my mind. Your love and snacks kept me alive.

To Ana and Emilio – thank you for welcoming me like family and for your support while I was in Europe. Having an extra set of parents rooting for me (and feeding me) made all the difference.

To Ittai Abraham – thank you for mentoring me during my internship at VMware Research. Your passion is infectious.

To Julian Loss – thank you for your guidance during my time at CISPA. Your support helped make research feel just a little less like chaos and a bit more like progress.

To my committee – Mike, Jeff, Jun, and Ashwin – thank you for your time, your feedback, and for not asking any trick questions (or at least not too many). I'm grateful for your thoughtful input and for helping me cross the finish line.

To Juan – thank you for listening to me rant about obscure topics you never asked to learn about, for tirelessly listening to hours of practice presentations, and for always believing in me.

To all of my collaborators that I haven't already mentioned - Naama, Gilad, Nibesh, Luca, Amey, Chen-Da - I have thoroughly enjoyed working with every one of you, and it has made me a better researcher and writer.

1. Introduction

Consensus enables a network of parties to come to agreement on a value despite the presence of faulty parties. Solving consensus has been an integral part of distributed computing over the past several decades, and has had various applications, such as distributed databases [20, 75, 77, 12, 18], large-scale cloud services [16, 61, 80], and most recently, blockchain systems [82, 47]. For any application, latency, the time it takes for all honest parties to decide, is an important metric when deciding whether to use a certain implementation in practice. To see why, consider the application of a blockchain system, such as the Bitcoin system, for payments. Blocks are created at a rate of one block per 10 minutes, meaning that a client must wait up to 10 minutes for their transaction to be included in a block. Contrast this to Visa, which enables payments that take seconds to complete. This has presented a huge barrier to the widespread adoption of blockchain systems for payments, and has resulted in the emergence of many solutions for increasing throughput and reducing latency [85, 78, 83]. For this reason, we are interested in designing consensus protocols with improved latency.

Another concern when choosing to use a certain consensus protocol in practice is the type of adversary that the protocol may tolerate. One question is whether the protocol tolerates a *crashing* adversary or a *Byzantine* adversary, which may deviate arbitrarily from the protocol specification. Another question is whether the adversary is *adaptive* or *static*. Unlike a static adversary, an adaptive adversary may update their strategy during the course of the protocol subject to constraints. An adaptive adversary could chose, for example, to delay messages or which parties to corrupt during the protocol execution as long as it stays within the corruption threshold. This concern is practically relevant, as parties could become corrupted with malware during the course of a protocol. A real concern with blockchain systems is that parties could be bribed to act in a malicious manner once their importance in the protocol is revealed. A *mobile adversary* is one that may change the set of corrupted parties during the course of the protocol, as long as the number of corrupted parties stays within the corruption threshold at any given point in time [87]. Another concern related to the ability of a protocol to tolerate change during its execution that is relevant for blockchain

systems is that the network should be dynamic: parties may join and leave the network. Traditional consensus protocols have operated in the permissioned model, which assumes that the network contains a fixed set of parties that all know each other and that the uncorrupted parties are always active. On the other hand, in the dynamically available model, parties may go online and offline freely and the total number of active participants is unknown [59]. The first protocol to satisfy the dynamic availability property was the Bitcoin protocol [71], and this property enables blockchain systems to continue making progress even when a large fraction of the network goes offline. For these reasons, we are interested in consensus protocols that are *efficient* in terms of their latency and which are secure considering changing network conditions that may be dictated by the adversary. All of the protocols in this work are *adaptively secure*, and we explore protocol design in the dynamically available setting.

There are various barriers to achieving consensus with better latency. Typically, consensus protocols proceed in a series of rounds in which the parties exchange messages (e.g. [11]). One way to think of such protocols is that the parties maintain an *estimate* of the value they believe to be the decision value of a majority, and by echoing their estimates to the other parties in a series of rounds, parties update their estimates and commit a value once they are confident about their estimate. In practice, these rounds of message passing result in increased latency. As such, one way to improve the latency of a consensus protocol is by reducing the number of message passing rounds required. In Chapters 2 and 3, we study consensus that is efficient in the number of message passing rounds and secure against an adaptive adversary that may corrupt parties and delay messages during the course of the protocol execution.

Another barrier to achieving low latency is the amount of time parties must wait to receive messages from the other parties. In certain scenarios, such as when parties may join and leave the network, the protocol may not both proceed at the speed of the network and maintain the properties required by consensus [59]. In such settings, parties must wait for the pessimistic assumption on the network delay to receive messages from other parties before proceeding to the next step of the protocol. In Chapter 4, we tackle consensus under

this setting, and present a protocol that achieves latency that is better than the pessimistic network delay despite this impossibility. We describe our contributions in each of these chapters in more detail below.

Efficient and Adaptively Secure Asynchronous Agreement via Binding Crusader Agreement. In Chapter 2, we tackle the problem of consensus in an *asynchronous* network against an *adaptive* adversary. That is, the adversary may delay messages for any finite amount of time and it may choose the delay and who to corrupt dynamically based on all the messages it has seen so far. We study this problem for both *crashing* adversaries (which cease participation in the protocol) and *Byzantine* adversaries (which may deviate arbitrarily from the protocol). Fischer, Lynch, and Paterson showed that any protocol that solves agreement in the presence of even one crash failure in the asynchronous model must have a non-terminating execution, and Ben and Or showed that *randomization* is the key to solving asynchronous agreement such that it terminates in a finite expected number of rounds [43, 9]. Still, proving security against *adaptive* adversaries under asynchrony has been notoriously difficult and error prone. In this work, we highlight the importance in asynchronous agreement protocol design of obtaining a property we call *binding*, and we use this insight to create simple frameworks for asynchronous agreement under various settings that are *easy* to prove correct and *efficient* in the number of communication rounds required for termination.

On the Round Complexity of Asynchronous Crusader Agreement. In Chapter 3, we work toward understanding the minimum number of rounds of communication required to achieve consensus under asynchrony. We build on the insights of Chapter 2, which showed a simple framework for asynchronous consensus using the primitive of *Binding Crusader Agreement (BCA)* and its variations. We show new lower bounds on the number of communication rounds required for asynchronous Crusader Agreement (CA) and BCA in both the information theoretic and authenticated settings. In doing so, we present a generic model for proving round complexity lower bounds in the asynchronous setting. In the settings in which our attempts to prove lower bounds on round complexity fail, we instead show new, tight, rather surprising round complexity upper bounds for Byzantine fault tolerant BCA

with and without a PKI setup.

Dynamically Available Consensus Under Generalized Synchrony. In Total-Order-Broadcast (TOB), consensus is continuously run to agree on a growing ledger of client transactions. In Chapter 4, we solve Byzantine fault-tolerant (BFT) TOB in the *dynamically available* setting, in which parties may go online or offline freely, and the total number of active participants is unknown. Existing dynamically available protocols rely on a synchronous network assumption, which means their latency remains tied to the pessimistic network delay, even when actual conditions are better. This raises the question of whether a dynamically available BFT TOB protocol can maintain security under synchrony while committing blocks at a rate closer to the actual network delay.

To answer this, we make two key contributions. First, we introduce generalized synchrony, a new network model that serves as an intermediate between synchrony and partial synchrony. This model defines k progressively increasing delays, with the highest delay Δ_k always holding. Participants maintain k logs, ensuring that logs created at higher delays are prefixes of those at lower delays, thereby preserving consistency. Second, we design the first dynamically available BFT TOB protocol that operates securely under generalized synchrony. A key challenge is committing blocks at a rate of $O(\Delta_{ideal})$ without prior knowledge of Δ_{ideal}, the smallest network delay that holds. Our protocol ensures that the network eventually stabilizes to commit at least one block per $O(\Delta_{ideal})$ time, achieving improved latency without compromising security.

2. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement

This work was done in collaboration with Ittai Abraham (VMware Research) and Naama Ben-David (VMware Research), with Sravya Yandamuri as lead author.

Reaching agreement in the presence of faults has been a cornerstone of distributed computing for over 40 years. In this paper we consider the *asynchronous* model with an *adaptive* adversary, where the adversary is allowed to delay any message between any two parties by any finite amount and can choose the delay and who to corrupt dynamically based on all the messages it has seen so far. The celebrated impossibility theorem of Fischer, Lynch, and Paterson (FLP) [41] states that any protocol that solves agreement in the presence of even one crash failure in the asynchronous model must have a non-terminating execution. The groundbreaking result of Ben-Or [9] shows that *free choice*, or *randomization*, is the key to solving Asynchronous Binary Agreement that terminates in a finite expected number of rounds.

Our main conceptual contribution is to highlight the importance in protocols design of obtaining a property we call *binding*. We believe protocols for asynchronous agreement algorithms against *adaptive* adversaries that are designed to obtain our binding property are modular and easy to prove correct.

Abstractly, a protocol obtains the binding property if no matter what the adversary does, it is forced to choose (bind to) in the present in a way that restricts all future outcomes of the protocol. As an example, we say a protocol obtains binding if there is a point in time τ (typically before any non-faulty party reveals their randomness) such that based on the execution until time τ, there is some value b (the value b is determined at time τ) such that any extension of this execution (no matter what the adversary does in the future), no non-faulty party outputs b.

Intuitively, the binding property helps protocols overcome the FLP impossibility by forcing the adaptive adversary to irrevocably choose (bind to) a value before knowing the random choices of the non-faulty parties. We believe that reasoning about the binding property explicitly, as a first order goal for algorithm design, greatly helps clarity and analysis. We note

that proving security against *adaptive* adversaries is notoriously difficult and error prone. Even in the synchronous model implementing the natural ideal functionality is non-trivial [52, 48].

We use the following standard definitions to capture randomness via the abstract notion of a *common coin*, first introduced by Rabin [73]. Roughly speaking, an ϵ-*good*, t-unpredictable common coin needs to have two properties: (1) *Unpredictability* the adversary cannot predict the coin value before at least $t+1$ parties start participating in the protocol; and (2) *Good*: for any value $v \in \{0,1\}$, with probability $\geqslant \epsilon$, all non-faulty parties output v. Many $O(1)$-good common coins are based on Verifiable Secret Sharing [40, 15, 2, 19]. A $1/2$-good common coin is called a *strong* common coin and can be implemented in the authenticated setting using secret sharing [74, 13]. A canonical example is simply having each party choose a local random coin. This is a 2^{-n}-good common coin, which we call the *local common coin*.

Our framework is based on a new abstraction that we call *Binding Crusader Agreement* (BCA). As in standard Crusader Agreement [31], this functionality is similar to agreement, but allows some parties to output a special $\perp$ value. More specifically, it guarantees (1) Validity: when all non-faulty parties have the same input, this is the only output; and (2) Agreement: no two honest parties output two distinct non-$\perp$ values. Additionally, as discussed above, we require the protocol to satisfy the *binding* property: that is, at the time t at which the first non-faulty party terminates, there is a value $v \in \{0,1\}$ such that no non-faulty party can output v in any extension of the execution. Inspired by Gradecast [40], we also extend BCA to *Graded-BCA*, where each party outputs a value and a grade $g \in \{0,1,2\}$, intuitively corresponding to how confident they are in their output. Roughly speaking, in Graded-BCA, a grade of 2 implies that all parties have the same value. There are numerous results that use the graded approach [53, 10, 44]; our work is one of the few that use it explicitly for an asynchronous model.

We show that asynchronous agreement can be solved with rounds of BCA followed by a strong coin flip, or rounds of Graded-BCA followed by a flip of an ϵ-good coin. We treat this as a *framework* which can solve many variants of agreement, including crash-tolerant,

information-theoretic, and computational, depending on the implementation and guarantees that the plugged in (Graded) BCA provides. We then present several implementations of (Graded) BCA, and by optimizing those, arrive at algorithms for agreement that match or improve upon the state of the art algorithms in terms of expected communication rounds to termination.

We note that in this paper we consider the *worst case* expected number of rounds until *termination*. This includes rounds after parties commit a value during which they cannot yet terminate the protocol (this accounting is not always included in the literature). Furthermore, we note that optimizations for 'optimistic' runs with synchrony or in which all non-faulty parties have the same input, are useful in practice, but are not the topic of this paper.

2.0.1 Previous Approaches

In this section, we present an overview of Asynchronous Byzantine Agreement protocols of the past.

In [70], Mostéfaoui, Moumen, and Raynal present a protocol for Asynchronous Binary Byzantine Agreement with $O(n^2)$ messages that has optimal resilience of $t < \frac{1}{3}$ Byzantine processes. The protocol makes use of a perfect common coin as defined by Rabin in [73]. Tholoniat and Gramoli subsequently observe in [79] that the ABA protocol of [70] has a liveness violation. In order to terminate, the protocol requires a round that each party begins with the same value v. To achieve such a round, the protocol uses the common coin. At the end of each round, a party adopts the value of the random coin or a value $v \in \{0, 1\}$, such that all parties that don't adopt the coin value adopt v. Since the random coin is perfect, if all non-faulty parties adopt the value of the coin in a given round, or the coin matches the value v adopted by parties who don't adopt the coin value, then the conditions for termination are met. The flaw in the protocol is that an adversary can learn the value of the coin in a given round, and then choose v to be the opposite of the value of the coin in that round and force a non-faulty party to adopt v. If all non-faulty parties don't start the protocol with the same value, the adversary can force an infinite execution with this tactic.

We refer the reader to [79] and [14] for detailed descriptions of how the adversary ensures an infinite execution.

In Algorithm 4 of [14], Cachin and Zanolini present a protocol for ABA that also uses a perfect common coin and that they claim solves the issues of [70]. Their solution makes use of a strong t-unpredictable common coin as well as the assumption of a FIFO network. However, this protocol still has a liveness issue when non-faulty parties don't all start the protocol with the same value. We construct an example with a network of 4 parties; 3 non-faulty parties (X, Y, and S), and one Byzantine party B. Let X start with 0 and Y start with 1. The starting value of S can be either 0 or 1 (we will show the attack works in either case). X and Y both broadcast their starting values in VALUE messages to all of the other parties. B sends 1 to X in a VALUE message, and 0 to Y in a VALUE message. Upon receiving the VALUE messages for 1 from B and Y, X broadcasts a VALUE message containing 1 and similarly Y broadcasts a VALUE message containing 0 upon receiving the VALUE messages for 0 from X and B. Upon receiving its own VALUE message for 1, X *abv-deliver*'s 1, and likewise Y *abv-deliver*'s 0. X and Y send AUX messages for 1 and 0, respectively. B then sends 0 to X in a VALUE message and lets X receive Y's VALUE message for 0 (along with its own VALUE message for 0), and X *abv-deliver*'s 0. Similarly, Y *abv-deliver*'s 1 after receiving the VALUE messages for 1 from itself, X and B. X and Y now send their second AUX messages for 0 and 1, respectively. B sends AUX messages for 0 and 1 to both X and Y. X and Y now satisfy the condition on line 30 in Algorithm 4 of [14], and invoke *release-coin*. X and Y receive each other's *release-coin* invocation and learn the value of the coin (as does the adversary). The coin is revealed to be s, and X and Y meet the condition on line 33 with $B = \{0, 1\}$ and propose s in the next round.

We now show how S, to whom all messages for this round have been delayed thus far, can adopt $1 - s$ for the next round without violating FIFO. Without loss of generality, assume that $s = 0$. Then the adversary lets S receive Y's VALUE message for 1. In addition, B sends a VALUE message for 1 to S. S then broadcasts a VALUE message for 1 (if it hasn't already due to 1 being its starting value). S then *abv-deliver*'s 1 and sends an AUX message for 1. The adversary also lets S receive X's VALUE messages for 0 and 1, as well as its AUX message

for 1. B also sends S an AUX message for 1. At this point, S invokes *release-coin*, receives B's *release-coin* message, and subsequently reaches line 33 with $B = \{1\}$. As a result, S starts the next round with 1. Notice that had s been 1, the adversary could have forced S to reach line 33 with $B = \{0\}$, and therefore start the next round with 0, using the same tactic. Also, this tactic worked regardless of S's starting value and did not violate FIFO. S received the first 3 messages sent by X in order and only the first message sent by Y. One way to make this protocol work would be to use a $2f$-unpredictable coin. In that case, S would only have learned the value of the coin after receiving all of the AUX messages of one of the other non-faulty parties. Since X and Y both sent AUX messages for both 0 and 1 prior to invoking *release-coin*, S would only satisfy the condition on line 33 with $B = \{0, 1\}$ and therefore would have also adopted the coin value. This solution would still require a FIFO network which, as we show in our work, is not a necessary assumption.

2.0.2 Our contributions: Crash failures

Asynchronous agreement can be solved while tolerating a minority of crash failures using randomization. This was first showed by Ben-Or in 1983 [9], but only against an oblivious adversary. Later, Aguilera and Toueg [6] showed that Ben-Or's protocol in fact works against an adaptive adversary as well. However, Aguilera and Toueg only considered a local coin, and showed expected termination in $O(2^{2n})$. Their results do not easily generalize to better coins. Using our framework, we obtain the following results.

Theorem 1. *Consider the problem of Asynchronous Binary Crash-tolerant Agreement with adaptive security, optimal resilience, asymptotically optimal message complexity for a network of n parties, where $t < \frac{n}{2}$ parties may be faulty*

1. *Given an ϵ-good t-unpredictable common coin, there exists a protocol that reaches termination in $3/\epsilon + 4$ communication rounds in expectation.*
2. *Given a strong t-unpredictable common coin, there exists a protocol that reaches termination in 7 communication rounds in expectation.*

Compared to Aguilera and Toueg [6]: (1) for the local common coin this is a quadratic improvement of the expected termination from $O(2^{2n})$ to $O(2^n)$; (2) for any ϵ-good common

Table 2.1: A comparison of previous results to ours in the crash-fault setting.

	Aguilera and Toueg[6]	Ours
Strong Coin	-	7
Weak Coin	-	$3/\epsilon$+4
Local Coin	$O(2^{2n})$	$O(2^n)$

Table 2.2: A comparison of our results in the Byzantine setting against previous results by coin strength. [0]

	[69]	[14]	[21]	Ours
Strong $t+1$	-	-	-	17
Strong $2t+1$	-	15	13	13
Weak $t+1$	$12/\epsilon+9$	-	$6/\epsilon+6$	$6/\epsilon+6$
Strong $2t+1$ w. Threshold Sigs	-	-	-	9

coin, when $\epsilon \gg 2^{-n}$ this provides a significantly better bound (exponential improvement for a $O(1)$-good common coin); (3) For a strong common coin, this provides the first result with adaptive security that are optimized for this setting.

2.0.3 Our contributions: Byzantine failures in the authenticated setting

In addition to our results in the crash fault setting, we present an efficient protocol for Asynchronous Binary Byzantine Agreement with threshold signatures. This result is relevant for the several papers, such as the work of Cachin, Kursawe and Shoup [13], as well as HoneyBadger [67], Beat [36], and DUMBO-MVBA [60], that use binary agreement and assume a DKG setup, and hence may benefit from this result for adaptive security and better round complexity. Other systems [23, 22] in the authenticated model could use this result to provide liveness in asynchrony.

Theorem 2. *Given a strong $2t$-unpredictable coin, and a Threshold Signature Setup, there exists a protocol solving Authenticated Asynchronous Binary Byzantine Agreement with adaptive security against a computationally bounded adversary, optimal resilience, and asymptotically optimal message complexity that reaches termination in 9 communication rounds in expectation.*

[0] To the best of our knowledge, [21] has not been peer reviewed.

2.0.4 Our contributions: Byzantine failures in the information theoretic setting

Ben-Or's [9] Byzantine resilient protocol does not have optimal resilience. Bracha's [11] protocol has optimal resilience but requires $O(n^3)$ message complexity, while the asymptotically optimal message complexity is $O(n^2)$ [32]. Mostefaoui, Moumen, and Reynal [70] solve Asynchronous Binary Agreement with the asymptotically optimal $O(n^2)$ message complexity and expected $O(1/\epsilon)$ rounds given an ϵ-good coin. This celebrated paper won the PODC 2014 best paper award, but it was later discovered that the protocol is not secure against adaptive adversaries. To remedy this problem, the JACM version of Mostefaoui, Moumen, and Reynal [69] had major protocol changes, more than doubling the round complexity. In particular, the worst case expected round complexity of [69] is $12/\epsilon + 9$ so even with a perfect coin the expected termination is 24+9=33 rounds. Cachin and Zanolini [14] suggest an improvement given a strong coin, but we show this improvement is not adaptively secure (liveness violation) when instantiated with a t-unpredictable strong coin.

Crain [21] in February 2020 presents improvements both for a strong coin and a weak coin. We note that these protocols, while not peer reviewed, have been used in other academic papers [28].

In this paper, we match Crain's expected rounds until termination through our framework by presenting efficient information theoretic Byzantine-tolerant implementations of (Graded-) BCA.

Theorem 3. *Consider the problem of Asynchronous Binary Byzantine Agreement with adaptive security, optimal resilience, asymptotically optimal message complexity for a network of n parties, where $t < \frac{n}{3}$ parties may be faulty*

1. *Given an ϵ-good t-unpredictable common coin, there exists a protocol that reaches termination in $6/\epsilon + 6$ communication rounds in expectation.*
2. *Given a strong t-unpredictable common coin, there exists a protocol that reaches termination in 17 communication rounds in expectation.*
3. *Given a strong $2t$-unpredictable common coin, there exists a protocol that reaches termination in 13 communication rounds in expectation.*

We believe that our approach is simpler than that of Crain, further highlighting the advantage of our agreement framework; it allows to match state-of-the-art algorithms while abstracting out key properties, making proofs modular, and generalizing to other settings as well.

2.1 Model

We consider an asynchronous message passing system with n parties, up to t of which may fail. That is, processes may communicate with one another by sending messages on links that do not drop messages, but may take an arbitrarily long amount of time to be received (all messages do eventually reach their destination). We consider two types of failures; *crash* failures, in which a party stops executing, and *Byzantine* failures, in which a party may act arbitrarily. Any party that does not fail in either manner is called *non-faulty*. An *adaptive adversary* controls which parties fail, and the amount of delay each message experiences. The adversary does not know the future, but does have access to all parties' private state. In particular, the adversary knows the value of random coin flips at the moment that this value is revealed to the party that flipped the coin.

We consider the following abstraction for random coins.

Definition 4 (ϵ-Good , d-Unpredictable Coin)**.** *An* ϵ-good d-unpredictable coin *provides an* access *primitive that takes no arguments and outputs either* 0 *or* 1 *with the following properties:*

- *The output value for all parties is completely unpredictable until at least* $d + 1$ *parties have accessed the coin.*
- *With probability at least* ϵ*, all non-faulty parties get* 0*, and with probability at least* ϵ*, all non-faulty parties get* 1*.*

An ϵ*-good and* d*-unpredictable* common coin has two properties: (1) *Unpredictability* the adversary cannot predict the coin value at long as at most d parties started the protocol; (2) *Good*: for any value $v \in \{0, 1\}$, all non-faulty parties output v with probability $\geqslant \epsilon$. If a coin is $1/2$-good, we call it a *strong coin*. Many $O(1)$-good common coins are based on Verifiable Secret Sharing [40, 15, 2]. A *strong* common coin and can be implemented in the authenticated setting using secret sharing [74, 13]. Building coins of various goodness has

been studied in other works and is not the topic of this paper. Instead, we use ϵ-good coins as a black box. Since many strong common coins are based on polynomial secret sharing, we will interchangeably say d-unpredictable and degree d.

We are interested in solving binary agreement problems in this setting. In particular, we consider the following two standard variants of agreement, which correspond to a setting with Byzantine failures and a setting with only crash failures, respectively.

Definition 5 (Asynchronous Byzantine Agreement (ABA))**.** *An asynchronous Byzantine agreement protocol among n parties is one in which each party receives an input and outputs a value v and has the following guarantees:*

- ***(Agreement)*** *If a non-faulty party commits v and another non-faulty party commits v', $v = v'$.*
- ***(Validity)*** *If all non-faulty parties receive the same value v as input, all non-faulty parties commit v.*
- ***(Termination)*** *All non-faulty parties eventually commit and then eventually terminate with probability 1.*

Definition 6 (Asynchronous Crash Agreement (ACA))**.** *A crash consensus protocol among n parties is one in which each party receives an input and outputs a value v and has the following guarantees:*

- ***(Agreement)*** *If a party commits v and another party commits v', $v = v'$.*
- ***(Weak Validity)*** *If all parties receive the same input value v, all non-faulty parties commit v.*
- ***(Termination)*** *All non-faulty parties eventually commit and then eventually terminate with probability 1.*

ACA is sometimes referred to as crash-tolerant consensus, uniform consensus, or simply *consensus* in the literature. The definition for ACA differs from that of ABA in that the agreement property of ACA is stronger as it must apply for *any two parties* that output values. The validity property of ACA is weaker in that it only applies if *all parties* start the protocol with the same value. These differences in definition are standard and reflect the nature of failures assumed for these two problems; generally, it is assumed that parties

executing ACA only fail by crashing, whereas parties executing ABA can fail arbitrarily. The weaker validity property is needed to allow ACA to be solvable with $2f + 1$ parties. In this paper, when referring generally to either one of these agreement problems, we say *Asynchronous Agreement (AA)*. We note that in this paper, we only consider Asynchronous Agreement with binary input. We therefore drop the 'binary' qualifier in the rest of the paper.

2.1.1 Notations

For the reader's convenience, we provide here all the notations we use to refer to different problem variants throughout this paper, including ones defined later on. The notations are also introduced in the first place that uses them.

We use the acronym $ABA_{\frac{1}{2}}$ to refer to Asynchronous Byzantine Agreement with a strong coin. ABA_{ϵ} refers to Asynchronous Byzantine Agreement with an ϵ-good coin. BCA_{Byz} refers to Binding Crusader Agreement for a Byzantine adversary, while BCA_{Crash} refers to Binding Crusader Agreement for a crash adversary. $BCA_{Byz,TSig}$ refers to Binding Crusader Agreement with threshold signatures for a Byzantine adversary. $EVBCA_{Byz}$, $EVGBCA_{Byz}$, and $EVBCA_{Byz,TSig}$ refer to the Externally Valid BCA, Externally Valid Graded BCA, and Externally Valid BCA with threshold signatures protocols in the Byzantine setting. The notation $ABA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ refers to the $ABA_{\frac{1}{2}}$ protocol with $BCA_{Byz,TSig}$ plugged in for BCA.

2.2 A Framework for Randomized Agreement

In Section 2.0.1, we describe Asynchronous Byzantine Agreement algorithms of the past that had flawed proofs of termination. They all follow a common pattern; parties execute rounds in which they try to decide a value, and if they cannot decide, they flip a coin to use as their input value in the next round. The counter examples to these algorithms similarly all stem from the same core problem: after the coin value is revealed, the adversary can still influence some parties to choose either value, and in particular, can force some of them not to adopt the coin value for the next round.

Our main contribution in this paper is capturing this common algorithmic pattern in a

simple unifying framework, which proceeds in rounds of *crusader agreement* followed by a coin flip. Crusader agreement [31] allows parties to decide $\perp$ instead of one of the input values if not all parties received the same input. Intuitively, in our framework a party adopts the value of the coin in round r if and only if the decision of its rth crusader agreement instance was $\perp$.

To make this framework solve asynchronous agreement against an adaptive adversary, we introduce a new property, called *binding*, to crusader agreement. The binding property dictates that by the time at which the first non-faulty party decides in an instance of crusader agreement, there is some value that no non-faulty party can decide. Thus, in the context of our agreement framework, this means that the adversary must 'bind' to some value before it sees the value of the coin in each round. Binding Crusader Agreement (`BCA`), is thus defined as follows.

Definition 7 (Binding Crusader Agreement (`BCA`))**.** *A Binding Crusader Agreement protocol between n parties has the following guarantees:*

> ***(Agreement)*** *If two non-faulty parties decide values x and y, then either $x = y$ or at least one of the values is $\perp$.*
>
> ***(Validity)*** *If all non-faulty parties have the same input, then this is the only possible decision for non-faulty parties.*
>
> ***(Termination)*** *All non-faulty parties eventually decide and then eventually terminate.*
>
> ***(Binding)*** *Let time τ be the first time such that there is a party that is non-faulty at time τ and decides at time τ. At time τ, there is a value $b \in \{0, 1\}$ such that no non-faulty party decides $1 - b$ in any extension of this execution.*

Note that the binding property is only interesting in the case that the non-faulty party referred to in the definition decided $\perp$. Otherwise, it trivially follows from agreement. To be able to implement `BCA` with an optimal tolerance to crash faults, we must weaken its validity property to match that of `ACA`. In particular, we must use the following validity property:

- **(Weak Validity)** If all parties have the same input v, then all non-faulty parties decide v.

We refer to `BCA` with this weak validity property as `BCA-Crash`, and to the one with the stronger validity property as `BCA-Byz`. That is, if any party, even a faulty one, has input value v to an instance of `BCA-Crash`, the protocol is allowed to decide v. This is known as a *weak validity* property, and has been shown to be necessary for solving agreement in an asynchronous setting with $n \geqslant 2t + 1$ parties, even under crash faults [8, 66]. The proof of that result can easily be modified to apply to BCA as well.

The full definition of `BCA-Crash` is as follows.

Definition 8 (Binding Crusader Agreement (BCA_{Crash}))**.** *A Binding Crusader Agreement protocol between n parties has the following guarantees:*

> ***(Agreement)*** *If two parties decide values x and y, then either $x = y$ or at least one of the values is $\perp$.*
>
> ***(Validity)*** *If all parties have the same input, then this is the only possible decision.*
>
> ***(Termination)*** *All non-faulty parties eventually decide.*
>
> ***(Binding)*** *Let time t be the the first time at which a party decides; at time t there is a value $b \in \{0, 1\}$ such that no party decides $1 - b$ in any extension of this execution.*

Other changes include requiring that *all* parties that terminate satisfy agreement (not just the non-faulty ones). Similarly, binding applies to all parties, including faulty ones.

Any solution to `BCA` (of either kind) can then be plugged into our framework to yield a solution to asynchronous agreement. The properties of the `BCA` implementation directly translate to properties of the resulting agreement protocol. For example, the agreement protocol inherits its fault tolerance and validity from its `BCA`. Thus, the same framework captures both crash-tolerant and Byzantine-tolerant agreement.

We present two versions of the framework depending on the strength of the coin that is used. The first uses `BCA` as was explained above, but requires a strong coin. That is, in each round, the coin must give the same value to all parties. To handle a weaker coin guarantee, we replace binding crusader agreement with its *graded* version. Graded crusader agreement was introduced by Feldman and Micali [40], and has parties output, along with their decision value, a *grade* that intuitively indicates how sure they are of their decision

value. The definition of GBCA is as follows.

Definition 9 (Graded Binding Crusader Agreement (GBCA))**.** *In a Graded Binding Crusader Agreement protocol, each party decides a value from a set of 5 ordered buckets:* 0 *grade* 2, 0 *grade* 1, $\perp$ *grade* 0, 1 *grade* 1, 1 *grade* 2 *and has the following guarantees:*

> ***(Graded Agreement)*** *If a non-faulty party decides v grade 2 or v grade 1, no non-faulty party decides $1 - v$ grade 1 or 2. Further, if a non-faulty party decides v grade 2, no non-faulty party decides $\perp$ grade 0.*
>
> ***(Validity)*** *If all non-faulty parties have the same input v, then all non-faulty parties decide v grade 2.*
>
> ***(Termination)*** *All non-faulty parties eventually decide and then eventually terminate.*
>
> ***(Graded Binding)*** *Let time τ be the first time such that there is a party that is non-faulty at time τ and decides at time τ. At time τ, there is a value $b \in \{0, 1\}$ such that no non-faulty party decides $1 - b$ grade 1 or 2 in any extension of this execution.*

We call this version of GBCA *GBCA-Byz*, and define GBCA-Crash by weakening the validity property as with the non-graded version. The full definition of GBCA-Crash is as follows.

Definition 10 (Graded Binding Crusader Agreement ($GBCA_{Crash}$))**.** *A Graded Binding Crusader Agreement protocol between n parties is one in which each party decides a value from a set of 5 ordered buckets:* 0 *grade* 2, 0 *grade* 1, $\perp$ *grade* 0, 1 *grade* 1, 1 *grade* 2, *and has the following guarantees:*

> ***(Graded Agreement)*** *If a party decides v grade 1 or 2, no party decides v' grade 1 or 2 such that $v' = 1 - v$. Further, if a party decides v grade 2, no party decides $\perp$ grade 0.*
>
> ***(Validity)*** *If all parties have the same input v, then v grade 2 is the only possible decision.*
>
> ***(Termination)*** *All non-faulty parties eventually decide.*
>
> ***(Graded Binding)*** *Let time t be the the first time at which a party decides; at time t there is a value $b \in \{0, 1\}$ such that no party decides $1 - b$ grade 1 or 2 in any extension of this execution.*

With graded BCA, parties in our framework only commit in their agreement instance if the decision of their GBCA was grade 2 (high confidence), and only adopt the coin if their

decision was grade 0 (equivalently, if they decide $\perp$). If a party decides a non-$\perp$ value v with grade 1 in round r of the framework, it will adopt v as the input for the next round, but will not decide yet. Of course, this version of the protocol works with a strong coin as well, but as we show later on, `BCA` can be solved more efficiently than its graded counterpart, and therefore our strong-coin version of the framework allows for more efficient solutions.

In the rest of this section, we present pseudocode for the two versions of our framework and prove that the Byzantine and Crash tolerant versions of (graded) `BCA` can be pluggged in to yield solutions to `ABA` and `ACA` respectively.

A note on termination. In our framework, when a party commits a value for the asynchronous agreement protocol, it continues participating until all others commit as well. Achieving an actual termination point can be done by having every non-faulty party broadcasting a special "commited" message upon deciding a value v. In the crash setting, a party can commit v once it receives such a message, broadcast the message, and terminate. In the Byzantine setting, once a party receives $t+1$ such messages, they can commit v and broadcast the same message. Upon receiving $2t+1$ such messages, a party can terminate. Therefore, in this section we simply measure the number of high level `BCA`-coin rounds until all parties commit. In later sections, we measure *broadcasts*, i.e., the number of communication steps required to terminate in an implementation of `BCA`. There, we report one extra broadcast to account for the commitment broadcast. Thus, in all our theorems we report rounds until actual termination (not just until commitment).

2.2.1 Asynchronous Agreement with a Strong Coin

Input: x
1: $r = 0$, $est = x$
2: **while** *true*
3: $r += 1$
4: $val = \texttt{BCA}(est)$
5: $c = \texttt{CommonCoin}()$
6: **if** $val \neq \perp$ and $c == val$ **then** commit val, $est = val$
7: **else if** $val \neq \perp$ **then** $est = val$
8: **else** $est = c$

FIGURE 2.1: Asynchronous Agreement with a Strong Coin

Our framework for solving asynchronous agreement with a strong coin is presented in

Figure 2.1. As described above, it proceeds in rounds of BCA followed by a coin flip. A party that decides $\perp$ in its BCA instance of round r adopts the coin value of r as the input to the BCA instance in round $r+1$. A party that decides a non-$\perp$ value v in round r will not adopt the coin value. Instead, if the coin value is also v, then it commits v. Otherwise, it uses v as its input to the BCA of round $r+1$.

Intuitively, a party that commits a value v in round r knows that it is safe to do so since all other parties will either have decide v in r as well, and therefore will have committed in round r as well, or they will have seen that the value of the coin is v, and will therefore adopt it in the next round. All non-faulty parties are therefore guaranteed to execute the next round of BCA with input v, and are therefore guaranteed to decide that value until they eventually commit it.

In the rest of this subsection, we show that by plugging in a Byzantine-tolerant version of BCA into Figure 2.1, we get a solution to ABA, and by plugging in a crash-tolerant BCA implementation, we get a solution to ACA. We note that ACA has a weaker validity property than ABA, and accordingly we define a weak validity version of BCA to allow for more fault tolerant implementations in the case of crash faults.

We now prove that this framework is correct. Theorem 11 formalizes this statement.

Theorem 11. *If we plug an implementation of BCA-Byz into BCA, and CommonCoin is a correct implementation of a strong coin, then Figure 2.1 solves Asynchronous Byzantine Agreement in any system in which $n \geqslant 3t+1$ and terminates in 4 rounds in expectation against an adaptive adversary.*

To prove Theorem 11, we start with a useful lemma.

Lemma 12. *If all non-faulty parties start a round r with the same estimate value $est = v$, then all non-faulty parties will commit v within a constant number of rounds.*

Proof. By the validity of BCA, since every non-faulty party inputs v to the instance of BCA, all non-faulty parties decide v. If $c = v$, all non-faulty commit v in this round. Otherwise, all non-faulty parties set $est = v$ and start the next round. This continues for each round

until we reach a round in which $c = v$, which happens in an expected constant number of rounds from round r. □

Using Lemma 12, it is now easy to show the correctness of the $AA_{\frac{1}{2}}$ protocol (Figure 2.1).

Proof of Theorem 11. We prove that each of the properties of asynchronous Byzantine agreement is satisfied.

Agreement. Note that a non-faulty party p only commits a value in a round r if the coin agreed with this value and it was the decision of p's `BCA` in round r. Let p be the first non-faulty party that commits, let r be the round in which it commits, and let v be its committed value. Note that in round r, no other value can be commited, since the commitment value is always the same as the coin of that round, and the coin is strong, so for any round r, all parties have the same coin value in r. Recall that `BCA` ensures that if a non-faulty party decides v from `BCA`, every other non-faulty party decides either $\perp$ or v from that instance of `BCA`. Since the coin value was v in r, all non-faulty parties must have started round $r + 1$ with $est = v$. Therefore, by Lemma 12, every other correct party q must also decide v.

Validity. If all non-faulty parties have the same input, then they all start round 0 with the same estimate. Therefore, by Lemma 12, they will all decide on that input.

Termination. Note that if in any round, either all non-faulty parties decide $\perp$ in their `BCA`, or the coin is the same as the non-$\perp$ decision value of `BCA`, then the lemma applies in the next round. Furthermore, by the binding property of `BCA`, the adversary is bound to the non-$\perp$ decision value of the `BCA` in any round r (if there is one) by the time the first non-faulty party finishes its `BCA` in round r. In particular, this must happen before the coin value is revealed in any coin of degree t or larger. Therefore, in each round, there is at least a 50% chance that the `BCA` decision value is the same as the coin, or $\perp$. □

We now show that the framework presented in Figure 2.1 also works for the crash tolerant case with `BCA-Crash`. Theorem 11 almost immediately implies that the frameworks work for the crash-fault setting as well, but we must ensure that the weaker validity property of `BCA-Crash` does not break correctness.

Theorem 13. *If we plug an implementation of* `BCA-Crash` *into* `BCA`*, and* `CommonCoin` *is a correct implementation of a strong coin, then Figure 2.1 solves Asynchronous Crash Agreement in any system in which* $n \geqslant 2t + 1$ *and terminates in 4 rounds in expectation against an adaptive adversary.*

We now prove Theorem 13. To prove the theorem, we rely on a version of Lemma 12 that is weakened to match the weak validity property of BCA_{Crash}.

Lemma 14. *If all parties start the* BCA_{Crash} *of round r of Figure 2.1 with the same estimate value* $est = v$*, then all non-faulty parties commit v within a constant number of rounds.*

Proof. The proof is very similar to that of Lemma 12. By the validity of BCA_{Crash}, since every party inputs v to the instance of BCA_{Crash} in round r, all parties that complete the `BCA` in that round decide v. If $c = v$, all participating parties commit v in this round. Otherwise, all participating parties start the next round with $est = v$. This continues until we reach a round in which $c = v$, which happens in 2 rounds in expectation. □

We are now ready to prove Theorem 13. Its proof is very similar to that of Theorem 11, except that we use Lemma 14.

Proof of Theorem 13. We consider each property of crash-tolerant agreement separately.

Agreement. Note that the agreement argument of Theorem 11 does not rely on the validity property of `BCA` at all, and can use Lemma 14 instead of Lemma 12. Therefore, agreement holds.

Validity. If all parties have the same input, then they all start round 0 with the same estimate. Therefore, by Lemma 14, they will commit on that input.

Termination. The termination proof is also very similar to its counterpart in Theorem 11. In any round in which all parties decide $\perp$ or the coin value agrees with the non-$\perp$ decision, Lemma 14 applies. The binding property of BCA_{Crash} combined with the strong coin ensures that this happens with probability at least 50% in each round. Thus, combining this with the expected two rounds for termination in Lemma 14, termination happens within 4 rounds in expectation. □

2.2.2 Asynchronous Agreement for Weak Coin from Graded BCA

The framework presented in Figure 2.1 relies heavily on the common coin giving the same random value to each non-faulty party in a given round. If the coin were to sometimes give different values to different parties, Lemma 12 would no longer hold, which could lead to agreement breaking.

More concretely, the following bad scenario could occur: a non-faulty party x decides value v in its `BCA` in a round r, and gets v from its coin in round r as well, and therefore commits v. However, all other non-faulty processes decide $\perp$ in their `BCA` in round r and receive a coin value of $1-v$. In this case, it is possible that these non-faulty parties decide $1-v$ in the `BCA` of round $r+1$, and could potentially commit $1-v$ if their coin for round $r+1$ agrees with this value. In this way, agreement would be violated.

However, implementing a strong common coin requires cryptographic support [13]. Thus, it is important to ask whether we can solve `AA` using a similar framework, while only making use of a *weak coin*.

```
1: Input: x
2: v = x
3: while true
4:     val, grade = GBCA(v)
5:     c = WeakCoin()
6:     if val == ⊥ then v = c
7:     else v = val
8:     if grade = 2 then commit val
```

FIGURE 2.2: Asynchronous Agreement with a Weak Coin

In this subsection, we show that this is possible by tweaking the `BCA` abstraction. In particular, as described in the beginning of Section 2.2, we make use of a gradecast, or *graded crusader agreement* abstraction, as introduced by Feldman and Micali [40].

We now show how `GBCA` can be used to solve `AA` using a weak coin. Pseudocode for this algorithm is presented in Figure 2.2. The idea is simple; the algorithm proceeds in rounds in which `GBCA` is run, followed by a coin flip. Intuitively, the coin and the decision of this round's `GBCA` together inform the input for the next round's `GBCA`. However, the coin is only ever adopted if the decision of `GBCA` was $\perp$. Otherwise, if the decision was some non-$\perp$ value

v, regardless of the grade, v will be this party's input for the next round. Furthermore, if the decision was grade 2, then the party commits v in this round. The key insight is that if a party decides grade 2 v and commits, then by graded agreement all others will have v as input for their next round, and will therefore all commit in the next round by validity. We formalize this argument by proving that plugging in implementations of `GBCA-Byz` and `GBCA-Crash` into Figure 2.2 yields solutions to `ABA` and `ACA` respectively regardless of the strength of the coin.

Theorem 15. *If we plug an implementation of `GBCA-Byz` into `GBCA`, and `WeakCoin` is a correct implementation of an ϵ-good coin, then Figure 2.2 solves Asynchronous Byzantine Agreement in any system in which $n \geqslant 3t + 1$ and terminates in $1 + 1/\epsilon$ expected rounds against an adaptive adversary.*

Theorem 16. *If we plug an implementation of `GBCA-Crash` into `GBCA`, and `WeakCoin` is a correct implementation of an ϵ-good coin, then Figure 2.2 solves Asynchronous Crash Agreement in any system in which $n \geqslant 2t + 1$ and terminates in $1 + 1/\epsilon$ expected rounds against an adaptive adversary.*

We now prove Theorem 15. We begin with a lemma very similar to Lemma 12, but even stronger.

Lemma 17. *If all non-faulty parties start a round r of Figure 2.2 with the same estimate value $est = v$, then all non-faulty parties commit v in round r.*

Proof. By the validity of `GBCA`, if all non-faulty parties start their `GBCA` with the same value v, they all decide v with grade 2. In that case, in Figure 2.2, they all commit v regardless of the value of the coin. □

Proof of Theorem 15. We prove that each of the properties of asynchronous Byzantine agreement is satisfied.

Agreement. Assume that two non-faulty parties x and y commit values v and v' respectively, where $v \neq v'$. If they commit in the same round, that means that x decided v grade 2 from its `GBCA` and y decided v' grade 2 in the same round of `GBCA`, violating graded agreement.

So, assume without loss of generality that x commits first, and let the round in which x commits be round r. By graded agreement of GBCA, since x had value v grade 2 in round r, all non-faulty parties must have decided v grade 1 or 2, and therefore by Figure 2.2, all non-faulty parties start round $r+1$ with v as their estimate. By Lemma 17, they all commit v – a contradiction to the assumption that y commited v'.

Validity. Immediate from Lemma 17.

Termination. Note that by the binding property of GBCA, by the time at which the first non-faulty party accesses the coin in a round r, there is some value, say v, that no non-faulty party can decide in round r (in any grade). Furthermore, by the definition of weak coin, there is a non-zero probability ϵ, that all non-faulty parties will get $1-v$ as their coin value in round r. If this happens, then all non-faulty parties start round $r+1$ with the same value $1-v$, and by Lemma 17, the protocol terminates in round $r+1$. Therefore, the protocol terminates in a number of rounds whose expectation is upper bounded by $1+1/\epsilon$ for an ϵ-good coin. □

We now show that the framework presented in Figure 2.2 works for crash-tolerant agreement as well.

For the proof, we rely on a weak validity version of Lemma 17.

Lemma 18. *If all parties start a round of Figure 2.2 with the same estimate value est $= v$ then all parties commit v in round r.*

Proof. By the weak validity property of $GBCA_{Crash}$, if all parties start their $GBCA_{Crash}$ instance with the same value v, they will all decide v with grade 2. In that case, in Figure 2.2, they all commit v regardless of the value of the coin. □

We are now ready to prove Theorem 16.

Proof of Theorem 16. The proof is similar to the proof of Theorem 15.

Agreement. The agreement proof of Theorem 15 does not rely on the validity property, and can use Lemma 18 instead of Lemma 17.

Validity. Immediate from Lemma 18.

Termination. The termination argument proceeds similarly to the termination argument of Theorem 15. By binding, by the time the first party accesses the coin in round r, there is some value v that cannot be decided by any party in the $GBCA_{Crash}$ of round r. If the coin gives the same value to all parties in round r, and that value agrees with the non-$\perp$ value they decide, then Lemma 18 applies. Therefore, the protocol terminates in a number of rounds whose expectation is upper bounded by $1 + 1/\epsilon$ for an ϵ-good coin. □

2.3 Implementing Binding Crusader Agreement

2.3.1 Asynchronous Crash Fault Tolerant Binding Crusader Agreement

We now present an protocol for weak validity BCA tolerating $t < n/2$ crash faults. The pseudocode is shown in Figure 2.3. To participate in the algorithm, a party p begins by sending its input value x to all parties in the system, labeled as a 'value message' with the tag `val`. It then waits to receive at least $n - t$ value messages, and checks whether they all contain the same value x. If so, p sends an echo message $\langle \text{echo}, id, x \rangle$ to all, and otherwise, it echos $\perp$, to indicate that it has witnessed at least two different initial values. p then waits to receive at least $n - t$ echo messages; if all of them again contain the same value x, p outputs x as its decision value. Otherwise, p outputs $\perp$.

Input: x
1: send $\langle \text{val}, id, x \rangle$ to all
2: **upon** receiving $\langle \text{val}, id, * \rangle$ messages from $n - t$ parties and not having sent an echo message:
3: **if** all of the messages contain the same value x, send $\langle \text{echo}, id, x \rangle$ to all
4: **else**, send $\langle \text{echo}, id, \perp \rangle$ to all
5: **upon** receiving $\langle \text{echo}, id, * \rangle$ messages from $n - t$ parties:
6: **if** all the messages contain the same value x, decide x
7: **else**, decide $\perp$

FIGURE 2.3: Asynchronous Binding Crusader Agreement for Crash Faults instance $id(BCA_{Crash})$

We now present the proof of correctness for the protocol in Figure 2.3, showing it is an implementation of weak validity `BCA` for crash faults with optimal fault tolerance.

Lemma 19. *The protocol in Figure 2.3 satisfies agreement.*

Proof. If a party outputs v, then they must have received at least $n - t$ echo messages containing v. Since parties send only a single echo message, by a simple quorum intersection

argument, no party can output $1 - v$. □

Lemma 20. *The protocol in Figure 2.3 satisfies weak validity.*

Proof. Assume that all parties start the protocol with value v and some party p outputs $1 - v$. p must have received $\langle \text{echo}, id, 1 - v \rangle$ messages from at least $n - t$ parties, all of whom received $\langle \text{val}, id, 1 - v \rangle$ messages from at least $n - t$ parties. However, this contradicts the fact that all of the parties started with v. □

Lemma 21. *The protocol in Figure 2.3 satisfies termination within 2 communication rounds.*

Proof. By inspecting the algorithm, it is easy to see that each party only ever sends at most 2 messages. Thus, if both **upon** clauses are eventually satisfied, the lemma holds.

Every non-faulty party sends a $\langle \text{val}, id, x \rangle$ message. Therefore, every non-faulty party receives a sufficient number of val messages to send an echo message. As a result, all non-faulty parties eventually decide. □

Lemma 22. *The protocol in Figure 2.3 satisfies binding.*

Proof. In order for a party to decide non-$\perp$ v, some party must have sent $\langle \text{echo}, id, v \rangle$. By quorum intersection, if a party sends $\langle \text{echo}, id, v \rangle$, then no party can send $\langle \text{echo}, id, 1 - v \rangle$. Since a party must receive $n - t$ $\langle \text{echo}, id, v \rangle$ to decide non-$\perp$ v, binding is achieved once the first party decides. □

Lemmas 19, 20, 21, and 22 together imply Theorem 23. Plugging this into Theorem 11, we get the proof for Theorem 24.

Theorem 23. *The protocol in Figure 2.3 implements weak validity* `BCA` *(*`BCA-Crash`*) tolerating* $t < \lceil n/2 \rceil$ *crash faults and all parties decide within 2 communication rounds.*

Theorem 23 and Theorem 13 together imply the following result.

Theorem 24. *There is a protocol that solves asynchronous crash-tolerant agreement in an expected 7 broadcasts in a system with* $n \geqslant 2t + 1$ *parties and a t-unpredictable strong common coin.*

Proof. Consider the $AA_{\frac{1}{2}}$-BCA_{Crash} protocol. The adversary has two strategies in a given round:

1. All parties output $\perp$ from BCA_{Crash}
2. Some parties output a non-$\perp$ value from BCA_{Crash}

If the adversary chooses the first strategy, then the parties adopt the same value and decide in the next round of $AA_{\frac{1}{2}}$ where the coin is equal to that value, so 3 rounds in expectation.

If the adversary chooses the second strategy, it takes in expectation two rounds of $AA_{\frac{1}{2}}$ for the coin to equal the non-$\perp$ decision value. Once this happens, some party must have commited.

The first strategy takes 3 rounds in expectation for a party to commit, while the second strategy takes 2 rounds in expectation. Once a party commits, they broadcast a "commit" message to all of the other parties and terminate. All of the other parties also broadcast this message and terminate once they receive it. Since there are 2 broadcasts per round of BCA_{Crash}, the protocol terminates in an expected 7 broadcasts. □

2.3.2 Asynchronous Byzantine Fault Tolerant Binding Crusader Agreement

We now present an implementation of `BCA` that tolerates Byzantine failures, and works as long as $n \geqslant 3t + 1$. The protocol is presented in Figure 2.4. In the pseudocode, we use **upon** to specify a clause that is triggered every time the specified condition is met, and **wait until** to indicate a clause that is only triggered *the first time* its condition is met.

The protocol proceeds as follows. Like in the crash-tolerant case, a party starts by sending its input value x to all other parties. In contrast to the crash-tolerant case though, here we have parties *amplify* values that they hear from at least one non-faulty party (i.e., $t + 1$ parties in total). The initial send of the value and the amplification are treated in the same way, and thus we call both types of messages an *echo* message.

Once $\langle$echo, $v\rangle$ for the same v is received from at least $n - t$ parties, a party p sends an *echo2* message with value v, and adds v to its $approvedVals$. Intuitively, p will never propagate or decide a value that isn't in its $approvedVals$ set. The $approvedVals$ set can later be updated to contain the other value $1 - v$ as well if $n - t$ $\langle$echo, id, $1 - v\rangle$ messages

are received. However, a non-faulty party only ever sends one echo2 message.

Intuitively, since each non-faulty party sends just one echo2 message, and that message always contains a non-$\perp$ value, we can think of echo2 messages as 'voting' for a value that we are sure all non-faulty parties will eventually hear and that some non-faulty party received as input. In the next phase of the protocol, a non-faulty party p aggregates the received votes and either amplifies a value v by sending $\langle$echo3, id, $v\rangle$ if it heard $n-t$ distinct parties vote for v with an $\langle$echo2, id, $v\rangle$ message, or sends $\langle$echo3, id, $\perp\rangle$ if it heard differing votes. An $\langle$echo3, id, $\perp\rangle$ message can also be sent if p has placed both v and $1-v$ in its $approvedVals$ set, meaning that eventually all parties will see and have both values in their $approvedVals$ sets as well.

Finally, p makes its decision by aggregating echo3 messages. If it receives at least $n-t$ echo3 messages with the same non-$\perp$ value v, it decides v. Otherwise, it waits until both v and $1-v$ are in its $approvedVals$ and it has received $n-t$ echo3 messages (regardless of their value) and then decides $\perp$. Intuitively, waiting to have both values in the $approvedVals$ helps validity; it ensures that the Byzantine parties can't force a decision of $\perp$ if all non-faulty parties started with the same value.

Input: x
1: $approvedVals = \{\}$
2: send $\langle$echo, id, $x\rangle$ to all
3: **upon** receiving $\langle$echo, id, $v\rangle$ for the same non-$\perp$ v from $t+1$ parties:
4: **if** haven't echoed v yet **then** send $\langle$echo, id, $v\rangle$ to all
5: **upon** receiving $\langle$echo, id, $v\rangle$ for the same non-$\perp$ v from $n-t$ parties:
6: $approvedVals = approvedVals \bigcup v$
7: **if** haven't sent any echo2 message yet **then** send $\langle$echo2, id, $v\rangle$ to all
8: **wait until**
9: (1) $|approvedVals| > 1$ or
10: (2) received $\langle$echo2, id, $v\rangle$ for the same non-$\perp$ v from $n-t$ parties
11: **if** (1) **then** send $\langle$echo3, id, $\perp\rangle$ to all
12: **else** send $\langle$echo3, id, $v\rangle$ to all
13: **wait until**
14: (1) $|approvedVals| > 1$ and received echo3 messages from at least $n-t$ parties or
15: (2) received $\langle$echo3, id, $v\rangle$ for the same non-$\perp$ v from at least $n-t$ parties
16: **if** (1) **then** decide $\perp$
17: **else** decide v

FIGURE 2.4: Asynchronous Binding Crusader Agreement for Byzantine Faults instance id (BCA_{Byz})

We now prove that the protocol in Figure 2.4 satisfies the properties stated in Definition 83. More specifically, we show the following theorem, which is implied from Lemmas 26, 27, 29 and 31.

Theorem 25. *The protocol in Figure 2.4 implements binding crusader agreement that terminates in at most 4 communication rounds and works in an asynchronous system with $n \geqslant 3t + 1$ parties where t parties could be Byzantine.*

Lemma 26. *The protocol in Figure 2.4 satisfies agreement.*

Proof. Assume, for a contradiction, that two non-faulty parties p_i and p_j decide non-$\perp$ values x_i and x_j respectively, such that $x_i \neq x_j$. In order to decide x_i, p_i must have received echo3 messages from $n - t$ different parties with the value x_i. Similarly, p_j must have received echo3 messages from $n - t$ different parties with the value x_j. By quorum intersection this cannot happen since each non-faulty party sends only one echo3 message. □

Lemma 27. *The protocol in Figure 2.4 satisfies validity.*

Proof. We prove validity in two parts. First, we prove that if all non-faulty parties start with value v, no non-faulty party outputs $\perp$. Assume, for a contradiction, that all non-faulty parties start with v and some non-faulty party p_i outputs $\perp$. Notice that the condition under which a party decides $\perp$ (line 16) requires that that party's $approvedVals$ set contains two distinct values. In order for a party to have two values in its $approvedVals$ set, it must receive $n - t$ copies of $\langle$echo, $b\rangle$ for both $b = 0$ and $b = 1$. A party only sends an amplification echo for a value $v' \neq v$ (line 4) if they received at least $t + 1$ messages $\langle$echo, id, $v'\rangle$. Therefore, some non-faulty party must have started with value $1 - v$, a contradiction.

Next, we prove that if all non-faulty parties start with value v, no non-faulty party decides $1 - v$. In order to decide $1 - v$, a non-faulty party must have $1 - v$ in its $approvedVals$ set (line 15). Again, this requires that some non-faulty party started with $1 - v$, a contradiction. □

Lemma 28. *If all non-faulty parties begin the protocol in Figure 2.4 and no non-faulty party terminates, the approvedVals sets of all non-faulty parties are eventually equal.*

Proof. A non-faulty party adds value v to its $approvedVals$ set after receiving $\langle$echo, id, $v\rangle$ messages from at least $n - t$ distinct parties, at least $n - 2t$ of which had to have been non-faulty parties. Therefore, if a non-faulty party adds v to its $approvedVals$ set, all non-faulty parties receive at least $t + 1$ $\langle$echo, id, $v\rangle$ messages and send $\langle$echo, id, $v\rangle$ themselves if they haven't already (line 4). As a result, all non-faulty parties receive $\langle$echo, id, $v\rangle$ messages from at least $n - t$ parties and add v to their $approvedVals$ sets. □

Lemma 29. *The protocol in Figure 2.4 satisfies termination within 4 communication rounds.*

Proof. Since we are in the binary agreement case, it is clear by the pseudocode that a party sends at most 4 messages; either one or two echo messages, one echo2 message and one echo3 message.

Also because we are considering the binary agreement case, there must be some value v such that at least $t + 1$ non-faulty parties start with v. Therefore, eventually all non-faulty parties send $\langle$echo, id, $v\rangle$. All non-faulty parties will then receive at least $n - t$ $\langle$echo, id, $v\rangle$ messages and send an echo2 message if they haven't already. In addition, each party will add v to its $approvedVals$ set.

In order for a non-faulty party to proceed in the protocol by sending an echo3 message, either the condition on Line 10 or Line 9 must be reached and satisfied. Note that non-faulty parties only send echo2 messages containing non-$\perp$ values. We consider two cases to prove that every non-faulty party eventually sends an echo3 message:

1. All non-faulty parties send echo2 messages for the same value v. Thus the condition on Line 10 is triggered and met once it receives echo2 messages from all non-faulty parties.
2. Some non-faulty party sends $\langle$echo2, id, $v\rangle$, and some non-faulty party sends $\langle$echo2, id, $1 - v\rangle$. By Lemma 28, the $approvedVals$ sets of all non-faulty parties eventually contain both v and $1 - v$. Therefore, eventually the condition on Line 9 will be met.

We have thus proved that all non-faulty parties send echo3 messages, and we must now prove that all non-faulty parties decide. We therefore must show that either the condition on Line 14 or the condition on Line 15 is reached and satisfied for all non-faulty parties that

have not yet decided. We consider two cases:

1. All non-faulty parties send $\langle$echo3, id, $v\rangle$ for the same non-$\perp$ value v. So clearly, Line 15 will be reached and satisfied.
2. Not all non-faulty parties send $\langle$echo3, id, $v\rangle$ for the same non-$\perp$ value v. If a non-faulty party sent $\langle$echo3, id, $\perp\rangle$, then its $approvedVals$ must have had more than one value in it. Furthermore, since any non-faulty party that sends $\langle$echo2, id, $v\rangle$ must have v in its $approvedVals$, and a non-faulty party must receive $\langle$echo2, id, $v\rangle$ from at least $t+1$ non-faulty parties before sending $\langle$echo3, id, $v\rangle$, any non-$\perp$ value that was sent in an echo3 message by a non-faulty party must be in some non-faulty party's $approvedVals$. By Lemma 28, eventually the $approvedVals$ sets of all non-faulty parties contain both 0 and 1. Since we've shown above that all non-faulty parties eventually send an echo3 message, the condition on Line 14 will eventually be satisfied.

Therefore, all non-faulty parties eventually decide. □

Lemma 30. *If a non-faulty party sends* $\langle$echo3, id, $v\rangle$*, where* v *is a non-bot value, no non-faulty party sends* $\langle$echo3, id, $1-v\rangle$*.*

Proof. If a non-faulty party sends $\langle$echo3, id, $v\rangle$ for a non-$\perp$ v they received $\langle$echo2, id, $v\rangle$ from at least $n-t$ parties. Since non-faulty parties send a single echo2 message, no non-faulty party can receive a sufficient number of $\langle$echo2, id, $1-v\rangle$ messages to send $\langle$echo3, id, $1-v\rangle$. □

Lemma 31. *The protocol in Figure 2.4 satisfies binding.*

Proof. Let p_i be the first non-faulty party to decide. p_i must have received at least $n-t$ echo3 messages, at least $t+1$ of which were from non-faulty parties. Consider these $t+1$ echo3 messages from non-faulty parties. If one of these messages contained a non-$\perp$ value v, then by Lemma 30, no non-faulty party sends $\langle$echo3, id, $1-v\rangle$ and therefore no non-faulty party can decide $1-v$. If all of the $t+1$ echo3 messages contain $\perp$, then no non-faulty party can receive enough echo3 messages with either non-$\perp$ values to decide either non-$\perp$ value. □

In Appendix 2.7.1 we show how, when plugging BCA_{Byz} into $AA_{\frac{1}{2}}$, we can use information from the $AA_{\frac{1}{2}}$ execution, such as the value of the coin in the previous round, to reduce the number of broadcasts per round. We define a new primitive based on `BCA` titled Externally Valid Binding Crusader Agreement (`EVBCA`) and present $EVBCA_{Byz}$, a modified version of the BCA_{Byz} protocol that implements `EVBCA`.

Theorem 32. *There is an protocol that solves asynchronous Byzantine fault tolerant agreement in expected 13 broadcasts in a system with $n \geqslant 3t + 1$ parties and a $2t$-unpredictable strong coin.*

Theorem 33. *There is an protocol that solves asynchronous Byzantine fault tolerant agreement in an expected 17 broadcasts in a system with $n \geqslant 3t + 1$ parties and a t-unpredictable strong coin.*

A note on space complexity. Note that as described, a non-faulty party may not terminate the BCA_{Byz} protocol until it has sent echo messages for both 0 and 1. This is because it may be required to continue participating in the protocol to ensure that all other non-faulty parties also terminate. Without ensuring that non-faulty parties eventually terminate each instance, sequentially running k instances requires each party to maintain $O(kn)$ local memory. In Section 2.5, we present a termination gadget to ensure that only the latest instance needs to be maintained.

2.4 Implementing Graded BCA

2.4.1 Asynchronous Crash Fault Tolerant GBCA

In Figure 2.5, we present a protocol for Asynchronous Graded Binding Crusader Agreement that withstands $n \geqslant 2t + 1$ crash faults. A party participating in this protocol first broadcasts their input, x, to all of the other in a val message. Upon receiving val messages from $n - t$ parties, a party broadcasts an $\langle$echo, $v\rangle$ message if they all contain the same value v and $\langle$echo, $\perp\rangle$ otherwise. Similarly, upon receiving echo messages from $n - t$ parties, a party broadcasts an $\langle$echo2, $v\rangle$ message if they all contain the same value v and $\langle$echo2, $\perp\rangle$ otherwise. Finally, once a party has received echo2 messages from $n - t$ parties, if they all contain the same non-$\perp$ value v or they all contain $\perp$, the party decides v (v grade 2) or $\perp$ respectively. If at least one of the echo2 messages contains non-$\perp$ value v and at least one of the echo2 messages contains contains a value other than v, the party decides $v\perp$ (v

grade 1). Notice that up until line 8, the protocol is nearly the same as that of BCA_{Crash} in Figure 2.3. The echo2 message a party sends corresponds to the value decided in the protocol for BCA_{Crash}.

Lemma 34. *The protocol in Figure 2.5 satisfies graded agreement.*

Proof. First, we prove that if a party outputs v grade 1 or 2, no party outputs v' grade 1 or 2 such that $v' = 1 - v$. If a party decides v (grade 1 or 2), they received at least one $\langle$echo2, id, $v\rangle$ message. In order to send $\langle$echo2, id, $v\rangle$, a party must have received $\langle$echo, id, $v\rangle$ messages from at least $n - t$ parties. Since a party sends an echo message for a single value v, the statement follows from a quorum intersection argument.

Next, we prove that if a party outputs v grade 2, no non-faulty party outputs $\perp$ grade 0. If a party outputs v grade 2, they received $\langle$echo2, id, $v\rangle$ messages from at least $n - t$ parties. Therefore, every party will see at least one $\langle$echo2, id, $v\rangle$ message. □

Lemma 35. *The protocol in Figure 2.5 satisfies termination within 3 communication rounds.*

Proof. It is clear from examining the protocol that parties broadcast at most 3 messages: a val message, an echo message, and an echo2 message.

Since there are at least $n - t$ non-faulty parties, every non-faulty party receives $\langle$val, id, *$\rangle$ messages from at least $n - t$ parties and sends an echo message. Every party therefore receives echo messages from at least $n - t$ parties, and all non-faulty parties send echo2 messages. Finally, every party receives at least $n - t$ echo2 messages, and all non-faulty parties decide a value. □

Lemma 36. *The protocol in Figure 2.5 satisfies validity.*

Proof. Assume that all parties start with v. We will prove that no party outputs $1 - v$ grade 1 or 2, v grade 1, or $\perp$ grade 0. In order to output such a value, a party must receive an echo2 message containing a value other than v. In order for a party to send such an echo2 message, a party must have sent an echo message for a value other than v. This means a party must have started with a value other than v, contradicting our original statement. Therefore, by termination, all the parties decide v grade 2. □

Lemma 37. *The protocol in Figure 2.5 satisfies binding.*

Proof. At the time at which the first party decides, at least $n - t$ parties must have sent echo2 messages. In order to output a value v grade 1 or 2, a party must receive at least one $\langle$echo2, id, $v\rangle$ message. Since a party sends only a single echo2 message with a non-$\perp$ value, by the time the first $n - t$ parties send echo2 messages, there must be a value $v' \in \{0, 1\}$ such that no party can output v' grade 1 or 2. □

Input: x
1: send $\langle$val, id, $x\rangle$ to all
2: **upon** receiving $\langle$val, id, $*\rangle$ messages from $n - t$ parties and not having sent an echo message:
3: **if** all of the messages contain the same value v **then** send $\langle$echo, id, $v\rangle$ to all
4: **else** send $\langle$echo, id, $\perp\rangle$ to all
5: **upon** receiving $\langle$echo, id, $*\rangle$ messages from $n - t$ parties and not having sent an echo2 message:
6: **if** all of the messages contain the same value v **then** send $\langle$echo2, id, $v\rangle$ to all
7: **else** send $\langle$echo2, id, $\perp\rangle$ to all
8: **upon** receiving $\langle$echo2, id, $*\rangle$ messages from $n - t$ parties:
9: **if** all of the messages contain the same value v **then** decide v, 2
10: **else if** $\geqslant 1$ of the messages contains v and $\geqslant 1$ of the messages contains a value other than v **then** decide v, 1
11: **else** decide $\perp$, 0

FIGURE 2.5: Asynchronous Graded Binding Crusader Agreement for Crash Faults instance id ($GBCA_{Crash}$)

Theorem 38. *The protocol in Figure 2.5 implements graded binding crusader agreement that terminates in at most 3 communication rounds and works in an asynchronous system with $n \geqslant 2t + 1$ parties where t parties could crash.*

Proof. The Theorem is implied by Lemmas 34, 35, 36 and 37. □

Theorem 39. *There is a protocol that solves asynchronous crash fault tolerant agreement in expected $3/\epsilon + 4$ broadcasts in a system with $n \geqslant 2t + 1$ parties and an ϵ-good coin.*

Proof. Consider the AA_ϵ-$GBCA_{Crash}$ protocol. If the value of the coin for all parties is equal to the non-$\perp$ decision value of the parties in that round, or they all output $\perp$, then Lemma 36 applies in the next round and all of the non-faulty parties commit. In expectation, it takes $1/\epsilon$ rounds for the parties to adopt the same value. Once they do, there is an additional round for them to decide that value, grade 2. There are 3 broadcasts per round and parties

send a final broadcast once they commit. This gives us a total of $3/\epsilon + 3 + 1 = 3/\epsilon + 4$ broadcasts. □

2.4.2 Asynchronous Byzantine Fault Tolerant Graded Binding Crusader Agreement

In this section, we present a protocol for Asynchronous Graded Binding Crusader Agreement that tolerates $t < \frac{n}{3}$ Byzantine faults. This protocol can be thought of as performing BCA_{Byz} in order to bind the adversary to a single non-$\perp$ value v that no non-faulty party can output, or to having $\perp$ be the only value that non-faulty parties can output. This gives us the graded binding property of Definition 84. Using the output of this first BCA_{Byz}, we then add an additional echo4 and echo5 round to get the graded agreement agreement property necessary for ABA with a weak coin. Graded agreement states that if some non-faulty party outputs v grade 2, all other non-faulty parties output v with grade 2 or 1. This ensures that if some non-faulty party decides v in the ABA protocol in a given round, all non-faulty parties start the next round with v and, due to the validity property, decide v in the next round of ABA.

As such, the protocol in Algorithm 2.6 is nearly identical to that of Figure 2.4 up through line 17. The echo4 message that a party sends corresponds to the value output from the protocol in Figure 2.4. Upon receiving echo4 messages from $2t + 1$ parties, a party sends an echo5 message for value v if they all contain the same non-$\perp$ value v or an echo5 message for $\perp$ if its $approvedVals$ set contains more than 1 value. Finally, a party decides based on the following criteria:

1. v grade 2 if it receives echo5 messages from $n - t$ parties for the same non-$\perp$ value v
2. v grade 1 if it receives echo5 messages from $n - t$ parties (at least one of which contains a non-$\perp$ value v), it receives $t + 1$ echo4 messages for v, and its $approvedVals$ set contains two values
3. $\perp$ grade 0 if it receives $n - t$ echo5 messages containing $\perp$ and its $approvedVals$ set contains two values.

By quorum intersection, it can't be the case that condition 1 is satisfied for one non-faulty party and condition 3 is satisfied for another non-faulty party in the same instance, ensuring

Input: x

1: $approvedVals = \{\}$
2: send $\langle$echo, id, $x\rangle$ to all
3: **upon** receiving $\langle$echo, id, $v\rangle$ for the same non-$\perp$ v from $t+1$ parties:
4: **if** haven't echoed v yet **then** send $\langle$echo, id, $v\rangle$ to all
5: **upon** receiving $\langle$echo, id, $v\rangle$ for the same non-$\perp$ v from $n-t$ parties:
6: $approvedVals = approvedVals \bigcup v$
7: **if** haven't sent an echo2 message yet **then** send $\langle$echo2, id, $v\rangle$ to all
8: **wait until**
9: (1) $\langle$echo2, id, $v\rangle$ has been received from $n-t$ parties for the same non-$\perp$ v **or**
10: (2) $\langle$echo2, id, $*\rangle$ messages have been received from $n-t$ parties and $|approvedVals| > 1$
11: **if** 1 **then** send $\langle$echo3, id, $v\rangle$ to all
12: **else** send $\langle$echo3, id, $\perp\rangle$ to all
13: **wait until**
14: (1) $\langle$echo3, id, $v\rangle$ has been received from $n-t$ parties for the same non-$\perp$ v **or**
15: (2) $\langle$echo3, id, $*\rangle$ messages have been received from $n-t$ parties and $|approvedVals| > 1$
16: **if** (1) **then** send $\langle$echo4, id, $v\rangle$ to all
17: **else** send $\langle$echo4, id, $\perp\rangle$ to all
18: **wait until**
19: (1) $\langle$echo4, id, $v\rangle$ has been received from $n-t$ parties for the same non-$\perp$ v **or**
20: (2) $\langle$echo4, id, $*\rangle$ messages have been received from $n-t$ distinct parties and $|approvedVals| > 1$
21: **if** (1) **then** send $\langle$echo5, id, $v\rangle$ to all
22: **else** send $\langle$echo5, id, $\perp\rangle$ to all
23: **wait until**
24: (1) $\langle$echo5, id, $v\rangle$ has been received from $n-t$ parties for the same non-$\perp$ v **or**
25: (2) $\langle$echo5, id, $*\rangle$ messages have been received from at least $n-t$ distinct parties, at least one of which is $\langle$echo5, id, $v\rangle$ s.t. v is non-$\perp$, $\langle$echo4, id, $v\rangle$ messages have been received from at least $t+1$ distinct parties, and $|approvedVals| > 1$ **or**
26: (3) $\langle$echo5, id, $\perp\rangle$ messages have been received from at least $n-t$ distinct parties and $|approvedVals| > 1$
27: **if** (1) **then** decide v, 2
28: **else if** (2) **then** decide v, 1
29: **else** decide $\perp$, 0

FIGURE 2.6: Asynchronous Graded Binding Crusader Agreement for Byzantine Faults instance id ($GBCA_{Byz}$)

graded agreement. In condition 2, checking that $t+1$ parties have sent $\langle$echo4, id, $v\rangle$ messages before outputting v grade 1 ensures that some non-faulty party has sent $\langle$echo4, id, $v\rangle$ so that the binding property of $AA_{\frac{1}{2}}$ holds. By checking that $approvedVals$ contains more than 1 value (conditions 2 and 3) prior to outputting v grade 1 or $\perp$ grade 0, we get validity.

We now prove that the protocol in Figure 2.6 implements Graded Binding Crusader Agreement. We first start with an important lemma.

Lemma 40. *If no non-faulty party terminates the protocol, then the approvedVals sets of all*

non-faulty parties are eventually equal.

Proof. A non-faulty party adds value v to its $approvedVals$ set if it receives $\langle$echo, id, $v\rangle$ messages from at least $n - t$ distinct parties. At least $t + 1$ of these messages had to have been from non-faulty parties, and therefore will be received by all non-faulty parties. As a result, all non-faulty parties send amplification echos for v (Lines 3-4), and all parties receive a sufficient number of echo messages for v to add v to their $approvedVals$ sets. □

Lemma 41. *The protocol in Figure 2.6 satisfies graded agreement.*

Proof. First we prove that if a non-faulty party outputs non-$\perp$ value v grade 1 or 2, then no non-faulty party outputs $1 - v$ grade 1 or 2. Assume that a non-faulty party outputs v grade 1. Then they must have seen $t + 1$ $\langle$echo4, id, $v\rangle$ messages, at least one of which was from a non-faulty party. The statement therefore follows from Lemma 26 and the fact that outputting v grade 2 requires more than one non-faulty party to send $\langle$echo4, id, $v\rangle$ (and likewise for $1 - v$).

Next, we prove that if a non-faulty party outputs non-$\perp$ value v grade 2, no non-faulty party outputs $\perp$ grade 0. In order to output v grade 2, a party must receive $\langle$echo5, id, $v\rangle$ messages from at least $n - t$ parties. By quorum intersection, and the fact that each non-faulty party broadcast a single echo5 message, no non-faulty party can receive the required $n - t$ $\langle$echo5, id, $\perp\rangle$ messages to decide $\perp$ grade 0. □

Lemma 42. *The protocol in Figure 2.6 satisfies validity.*

Proof. Assume all non-faulty parties start the protocol with v. By Lemma 27, all non-faulty parties send $\langle$echo4, id, $v\rangle$. Next, we show that no non-faulty party adds $1 - v$ to its $approvedVals$ set. To show this, we prove that no non-faulty party ever sends an $\langle$echo, id, $1 - v\rangle$ message. In order to send an $\langle$echo, id, $1 - v\rangle$ message, a non-faulty party must receive $\langle$echo, id, $1 - v\rangle$ from $t + 1$ parties. This cannot happen if all non-faulty parties start with v. Therefore, non-faulty parties will only send an echo5 message after receiving $n - t$ $\langle$echo4, id, $v\rangle$ messages, and will therefore all send $\langle$echo5, id, $v\rangle$. Since no non-faulty party adds $1 - v$ to its $approvedVals$, all non-faulty parties decide only upon receiving $n - t$ $\langle$echo5, id, $v\rangle$ messages and output v. □

Lemma 43. *The protocol in Figure 2.6 satisfies termination within 6 communication rounds.*

Proof. It is clear from examining the protocol that a party sends at most two echo messages and at most one echo2, one echo3, one echo4, and one echo5 message.

By Lemma 29, all non-faulty parties send echo4 messages. If all non-faulty parties send $\langle$echo4, id, $v\rangle$ for the same non-$\perp$ value v, then all non-faulty parties receive $\langle$echo4, id, $v\rangle$ messages from a sufficient number of parties to send $\langle$echo5, id, $v\rangle$. All non-faulty parties then receive $\langle$echo5, id, $v\rangle$ from a sufficient number of parties to decide v.

Next, we consider the case where all non-faulty parties do not send $\langle$echo4, id, $v\rangle$ for the same non-$\perp$ value v. Assume that this is the case. Then, by Lemma 26, the only other value non-faulty parties could have sent is $\langle$echo4, id, $\perp\rangle$. This non-faulty party must have its $|approvedVals| > 1$. By Lemma 40, $|approvedVals| > 1$ for all non-faulty parties eventually, and all non-faulty parties accept this echo4 message. As a result, all non-faulty parties send echo5 messages. If a non-faulty party sends $\langle$echo5, id, $v\rangle$, they received $\langle$echo4, id, $v\rangle$ from at least $n - t$ parties, and every parties is guaranteed to receive $\langle$echo4, id, $v\rangle$ from at least $t + 1$ parties. If a non-faulty party sends $\langle$echo5, id, $\perp\rangle$, again by Lemma 40, all non-faulty parties accept this echo5 message. As a result, all parties receive a sufficient number of valid echo5 messages and decide a value. □

Lemma 44. *The protocol in Figure 2.6 satisfies binding.*

Proof. In order for a non-faulty party to decide v or $v\perp$, at least one non-faulty party must send $\langle$echo4, id, $v\rangle$. The lemma therefore follows from Lemma 31. □

Theorem 45. *The protocol in Figure 2.6 implements graded binding crusader agreement that terminates in at most 6 communication rounds and works in an asynchronous system with $n \geqslant 3t + 1$ parties where t parties could be Byzantine.*

Proof. The Theorem is implied from Lemmas 41, 42, 43 and 44. □

Theorem 46. *There is a protocol that solves asynchronous Byzantine fault tolerant agreement in expected $6/\epsilon + 6$ broadcasts in a system with $n \geqslant 3t + 1$ parties, where t parties could be Byzantine, and an ϵ-good coin.*

Proof. Consider the AA_ϵ-$GBCA_{Byz}$ protocol. If the value of the coin for all non-faulty parties in a given round is equal to the non-$\perp$ decision value of the non-faulty parties in that round, or they all output $\perp$, then Lemma 42 applies in the next round and all the non-faulty parties commit. In expectation it takes $1/\epsilon$ rounds until this happens, and there are at most 6 broadcasts per round until it does. Since all non-faulty parties start the round in which they commit with the same value, there are 5 broadcasts in that round. There is one final broadcast after all the non-faulty parties decide to ensure termination. □

2.5 Garbage Collection

If a party outputs a non-$\perp$ value from an instance of BCA_{Byz}, they may need to still participate in the protocol after deciding a value to send an additional echo message if needed. Hence even after outputting a non-$\perp$ value, a party is required to maintain state and continue running the protocol to help others output a value. In this section, we present a termination gadget that can be added to BCA_{Byz} to ensure that all non-faulty parties eventually terminate the protocol. The same approach can be applied to the $GBCA_{Byz}$ protocol as well.

```
1: upon deciding ⊥:
2:     send ⟨output, id, ⊥⟩ to all
3:     terminate
4: upon deciding non-⊥ v:
5:     send ⟨output, id, v⟩ to all
6: upon receiving t + 1 ⟨output, id, v⟩ messages for non-⊥ v and not having decided:
7:     decide v
8:     send ⟨output, id, v⟩ to all
9: upon having sent ⟨echo, id, 0⟩ and ⟨echo, id, 1⟩ and having decided:
10:    terminate
11: upon receiving n − t ⟨output, id, v⟩ messages for non-⊥ v and having decided:
12:    terminate
```

FIGURE 2.7: Termination Gadget for BCA instance *id*

Lemma 47. *All non-faulty parties eventually decide a value in the BCA_{Byz}-Term protocol.*

Proof. The only case in which a non-faulty party terminates the *BCA_{Byz}-Term* protocol without sending echo messages for both non-$\perp$ values is if they received $n - t$ ⟨output, *id*,

$v\rangle$ messages, in which case all non-faulty parties receive a sufficient number of $\langle$output, id, $v\rangle$ messages to decide v (line 6). The Lemma follows from this fact and Lemma 29. □

Lemma 48. *All non-faulty parties eventually terminate the BCA_{Byz}-Term protocol.*

Proof. Note that if a non-faulty party sends echo messages for both non-$\perp$ values, they have no way of participating in the instance of the BCA_{Byz} protocol after deciding a value. The non-trivial aspect of the termination gadget is to ensure that non-faulty parties can terminate without sending echo messages for both non-$\perp$ values if needed. To prove that the protocol in Figure 2.7 achieves the required termination property, we distinguish between two cases:

1. All non-faulty parties output the same non-$\perp$ value v.
2. Some non-faulty party outputs $\perp$.

Note that by the agreement property of `BCA`, these are the only possible outcomes of an instance of `BCA`. We now prove that in both cases, all non-faulty parties eventually terminate the instance of `BCA`. In case 1, all non-faulty parties send $\langle$output, id, $v\rangle$ (line 5). All non-faulty parties eventually have the condition on line 11 satisfied, and they all terminate.

In case 2, the non-faulty party that output $\perp$ sends $\langle$output, id, $\perp\rangle$ (line 2) and terminates. Note that by Lemma 47, all non-faulty parties eventually decide a value. Since some non-faulty party decided $\perp$, at least $t+1$ non-faulty parties must have sent echo messages for both non-$\perp$ values. All non-faulty parties eventually receive these echo messages and, if they haven't already terminated, send echo messages for both non-$\perp$ values and terminate (line 9). □

2.6 Binding Crusader Agreement with Threshold Signatures

In this section, we present $BCA_{Byz,TSig}$, the `BCA` protocol that is used in $ABA_{\frac{1}{2}}$ to implement $ABA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$. $BCA_{Byz,TSig}$ tolerates $t < \frac{n}{3}$ Byzantine parties. We add two cryptographic assumptions in this model: digital signatures and threshold signatures. More specifically, we use a k out of l threshold signature scheme [74] i.e., k parties must participate in order to create a valid threshold signature. We use threshold signatures with two thresholds: for $k = t+1$ and $k = 2t+1$, both for $l = n$. pk_j refers to the public key of party

p_j. The interface for the threshold signature scheme is described in Appendix B. We now describe Algorithm 2, the protocol for $BCA_{Byz,TSig}$. First, we describe the two threshold signatures that are created in the protocol:

- $\sigma_{echo,id,v}$ is a threshold signature for $k = t + 1$ proving that a non-faulty party started instance id with value v.
- $\sigma_{echo3,id,v}$ is a threshold signature for $k = 2t + 1$ proving that $t + 1$ non-faulty parties sent echo3 messages for v, and thus, by binding, no non-faulty party can output $1 - v$ from this instance.

In this setting, we are able to do away with the amplification echo and $approvedVals$ set that were used in the information theoretic setting because $\sigma_{echo,id,v}$ is a sufficient proof that some non-faulty party started instance id with value v, and it can be sent along with any message.

A non-faulty party starts the protocol by broadcasting its starting value x in an echo message along with a threshold signature share on $\langle$echo, id, $x\rangle$. Upon receiving such messages from $t + 1$ parties containing valid threshold signature shares for the same value x, a party combines them into a threshold signature, $\sigma_{echo,id,v}$, and broadcasts it in an echo2 message for v. Upon receiving such an echo2 message for a value v and valid $\sigma_{echo,idv}$ for v, a party can broadcast the same echo2 message if it hasn't already sent an echo2 message. Upon receiving echo2 messages for values v' containing valid $\sigma_{echo,id,v'}$ from $2t + 1$ parties, a party broadcasts an echo3 message:

- If the $2t + 1$ echo2 messages are for the same value v, it broadcasts $\langle$echo3, v, $\sigma_{echo,id,v}$, $ts\rangle$, where ts is a threshold signature share on (echo, id, v).
- If the $2t + 1$ echo2 messages are not for the same value v, it broadcasts $\langle$echo3, $\perp$, $\{\sigma_{echo,id,v}, \sigma_{echo,id,1-v}\}$, $\perp\rangle$.

In both cases, the σ sent in the echo3 messages are from the echo2 messages that the party received. For each echo3 message that a party receives, it checks that it contains the necessary set of proofs and that it contains a valid threshold signature share if it is for a non-$\perp$ value. Upon receiving valid echo3 messages from $2t + 1$ parties, a party outputs $\perp$ if they are not all for the same value v, or v if they are all for the same value v. In addition, if

all the echo3 messages are for the same value v, a party combines the shares in the messages into a threshold signature $\sigma_{echo3,id,v}$ on (echo, id, v). In Appendix 2.7.2 , we show how to use this proof to implement $ABA_{\frac{1}{2}}$-$EVBCA_{Byz,TSig}$, an optimized version of $ABA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ that is able to reach termination with fewer broadcasts. The pseudocode and the proof of Theorem 49 are in Appendix B. Theorem 50 follows from the $EVBCA_{Byz,TSig}$ protocol, and the corresponding proofs can be found in Appendix 2.7.2.

Theorem 49. *The protocol in Figure 2 implements binding crusader agreement that terminates in at most 3 communication rounds and works in an asynchronous system with $n \geqslant 3t+1$ parties where t parties could be Byzantine.*

Theorem 50. *There is a protocol that solves asynchronous Byzantine fault tolerant agreement in expected 9 broadcasts in a system with $n \geqslant 3t+1$ parties, a $2t$-unpredictable strong coin, and a threshold signature setup.*

2.7 Externally Valid Binding Crusader Agreement

In previous sections, we presented protocols for Asynchronous Agreement and Binding Crusader Agreement in a decoupled fashion. Now that we have explained the simple framework for `AA` based on `BCA`, we show a series of modifications to make `AA` protocols terminate with fewer broadcasts. When using `BCA` within an `AA` protocol, we can take into account the context of the `AA` protocol within the `BCA` execution (such as the value of the coin of the previous round) to reduce the number of broadcasts in each round. In this section, we present these modifications and prove their correctness. These optimizations rely on the fact that in certain cases, the proof that a value or message is "valid" exists, and is guaranteed to be received by all parties. In these conditions, parties can omit sending certain messages for this value in the next round, or send certain messages early, since all parties are guaranteed to receive proof that this value or message is "valid" and act accordingly in that round.

To illustrate an example, consider the $AA_{\frac{1}{2}}$-BCA_{Byz} protocol. If a party has proof that some non-faulty party started round r of the protocol with 0 (i.e. by receiving $t+1$ echo messages for 0) and some non-faulty party started round r with 1, and the value of the coin in round r is 0, then the party knows that it is possible that some non-faulty party could

have decided $\perp$ within the BCA_{Byz} instance of round r and updated its *est* to 0, the value of the coin. In addition, it knows that 0 is a "safe value" for the subsequent round because no non-faulty party would have commited 1 in round as it is not the coin value. Non-faulty parties that adopt 0 in round r as a result of deciding $\perp$ from the instance of BCA_{Byz} can therefore omit sending echo messages for 0 in round $r+1$. Note that the existence of such a proof that 0 is a valid value *doesn't guarantee that a non-faulty party did output that value from* BCA_{Byz}; it only means that they *could have*. The modified protocols therefore no longer satisfy the original validity properties of `BCA` and `GBCA` as defined in Definition 83 and Definition 84. For this reason, we define a new validity property, *external validity*, that is a property of *Externally Valid Binding Crusader Agreement* (`EVGBCA`). External validity captures that a value is "externally valid" if a proof of its validity exists, even if no non-faulty party started with that value.

We first present a definition for Externally Valid Binding Crusader Agreement (`EVBCA`) and prove that when a protocol that implements `EVBCA` is plugged into $AA_{\frac{1}{2}}$ respectively, the result is a protocol implementing `AA`. In section 2.7.1, we should how to get Externally Valid Binding Crusader Agreement for Byzantine faults ($EVBCA_{Byz}$) from BCA_{Byz}. We then prove that $EVBCA_{Byz}$ implements `EVBCA`. In section 2.7.2, we show the same for Externally Valid Binding Crusader Agreement with threshold signatures for Byzantine faults ($EVBCA_{Byz,TSig}$) from $BCA_{Byz,TSig}$.

We first define Externally Valid Binding Crusader Agreement (`EVBCA`) with an external function called `Ext-Valid`() that returns whether a value is valid or not.

Definition 51. *(Externally Valid Binding Crusader Agreement (`EVBCA`)) An Externally Valid Binding Crusader Agreement protocol between n parties has the following guarantees:*

> ***(Agreement)*** *If two non-faulty parties output values x and y, then either $x = y$ or at least one of the values is $\perp$.*
>
> ***(External Validity)*** *If* `Ext-Valid`(v)=*true and* `Ext-Valid`$(1-v)$=$false$*, all non-faulty parties decide v.*
>
> ***(Termination)*** *All non-faulty parties eventually decide.*
>
> ***(Binding)*** *Let t be the first time at which a party that is non-faulty at the time of deciding*

decides. At time t there is a value $b \in \{0,1\}$ such that no non-faulty party decides $1-b$ in any extension of this execution.

Next, we define an externally valid value for the $AA_{\frac{1}{2}}$ in the context of the $AA_{\frac{1}{2}}$ protocol.

Definition 52. *(Externally Valid Value for $AA_{\frac{1}{2}}$) A value v is externally valid in round r of the $AA_{\frac{1}{2}}$ protocol if either of the following conditions are met:*

1. *A non-faulty party started round r of the protocol with v.*
2. *v was externally valid in round $r-1$ and no non-faulty party committed $1-v$ in round $r-1$.*

We now show that a protocol implementing `EVBCA` can be plugged into the $AA_{\frac{1}{2}}$ protocol (Figure 2.1) to solve `AA`.

Theorem 53. *If we plug an implementation of `EVBCA` into `BCA` and `CommonCoin` is a correct implementation of a strong coin, then the protocol in Figure 2.1 solves asynchronous Byzantine agreement in any system in which $n \geqslant 3t+1$ and terminates in 4 rounds in expectation against an adaptive adversary.*

Before proving Theorem 53, we start with a useful lemma.

Lemma 54. *If all non-faulty parties start round r with the same value $est = v$, and `Ext-Valid`$(1-v)=false$, then all non-faulty parties commit v within a constant number of rounds.*

Proof. By the external validity property of `EVBCA`, every non-faulty party outputs v from `EVBCA` and therefore sets $est = v$ in round r. Since no non-faulty party starts round $r+1$ with $1-v$ and $1-v$ was not externally valid in round r, `Ext-Valid`$(1-v)=false$ in round $r+1$. This continues for every round by the definition of external validity. In round $r' \geqslant r$ where v is equal to the value returned by `CommonCoin` for round r', all non-faulty parties commit v. In expectation, this happens in 2 rounds since `CommonCoin` has a 50% probability of returning v in each round. □

Proof of Theorem 53. We prove that each of the properties of asynchronous Byzantine agreement is satisfied.

Agreement. A non-faulty party p only commits a value in round r if the coin agreed with this value and it was the output of p's `EVBCA` in round r. Let p be the first non-faulty party that commits, let r be the round in which it commits, and let v be its commit value. Note that in round r no other value can be commited since the commit value is always the same as the coin of that round and the coin is strong. By the agreement property of `EVBCA`, every non-faulty party decides v or $\perp$ from `EVBCA`, and therefore all non-faulty parties set $est = v$ at the end of this round.

By the definition of an externally valid value, `Ext-Valid`$(1-v)$=$false$ in round $r+1$. By the external validity property of `EVBCA`, in subsequent rounds of $AA_{\frac{1}{2}}$ no non-faulty party ever decides $1-v$ or $\perp$ from `EVBCA` and $1-v$ never becomes externally valid. Therefore, no non-faulty party ever decides $1-v$ in round $r' > r$.

Validity. If all non-faulty parties start the protocol with v, then $1-v$ cannot be externally valid by the definition of an externally valid value. Therefore, by Lemma 54 all non-faulty parties will commit v.

Termination. Note that if in any round the coin value is equal to the non-$\perp$ decision value of non-faulty parties or all of the non-faulty parties output $\perp$, Lemma 54 applies in the next round. Furthermore, by the binding property of `EVBCA`, the adversary is bound to the non-$\perp$ value b that can possibly be output by a non-faulty party in a round r (if there is one), by the time the first non-faulty party finishes its `EVBCA` in round r. In particular, this must happen before the coin value is revealed in any coin of degree t or larger. Therefore, in each round, there is at least a 50% chance that the value b to which the adversary is bound will be the same as the coin or $\perp$. □

2.7.1 Implementing Externally Valid Binding Crusader Agreement

We now present modifications that can be applied to the BCA_{Byz} protocol of Figure 2.4 to obtain $EVBCA_{Byz}$, a protocol that implements `EVBCA`. When used with a $2t$ degree strong common coin in the $AA_{\frac{1}{2}}$ protocol of Figure 2.1, the $EVBCA_{Byz}$ requires at most 3 broadcasts per round, and a minimum of two broadcasts per round, for every round after the first round. We first present the definition of an externally valid value that is specific to $AA_{\frac{1}{2}}$-BCA_{Byz}.

We then prove that this definition satisfies the definition of an externally valid value for the $AA_{\frac{1}{2}}$ protocol in Definition 52. After that, we present the modifications to get $EVBCA_{Byz}$ from BCA_{Byz}. Finally, we prove that $EVBCA_{Byz}$ implements `EVBCA`.

Definition 55. *(Externally Valid Value for $AA_{\frac{1}{2}}$-BCA_{Byz}) A value v is **externally valid** for the in round r of the $AA_{\frac{1}{2}}$-BCA_{Byz} protocol if either of the following conditions are met:*

1. *v is eventually in the $approvedVals$ set of all non-faulty parties in round $r-1$ and v is the value of the common coin in round $r-1$.*
2. *Some non-faulty party starts round r of the protocol with value v.*

Lemma 56. *Externally Valid Value for $AA_{\frac{1}{2}}$-BCA_{Byz} (Definition 55) satisfies the definition of an Externally Valid Value for $AA_{\frac{1}{2}}$ in Definition 52.*

Proof. We use a proof by induction. Consider round 0 of the $AA_{\frac{1}{2}}$-BCA_{Byz} protocol. If all non-faulty parties start round 0 with value v, no non-faulty party adds $1-v$ to their $approvedVals$ set in round 0, and no non-faulty party decides $1-v$ or $\perp$ from the BCA_{Byz} of that round. By validity of BCA_{Byz}, no non-faulty party ever starts a round with $1-v$ or adds $1-v$ to their $approvedVals$ set. □

We now present the set of modifications that are applied to BCA_{Byz} to get the $EVBCA_{Byz}$ protocol.

1. In round r, p_i automatically adds v to its $approvedVals$ set if p_i's $approvedVals$ set of round $r-1$ contains v and v was the value of the common coin in round $r-1$.
2. Upon adding value v to its $approvedVals$ set in round r, if p_i did not already send an echo2 message in round r, p_i sends $\langle$echo2, r, $v\rangle$ to all parties.
3. If p_i outputs $\perp$ from the BCA_{Byz} of round r and the coin value in round r is c, p_i does not send an echo message in round $r+1$ and directly broadcasts an $\langle$echo2, $r+1$, $v\rangle$ message.
4. Upon outputting v from the BCA_{Byz} of round r, if the common coin of round r also equals v, then p_i automatically broadcasts the messages $\langle$echo2, $r+1$, $v\rangle$ and $\langle$echo3, $r+1$, $v\rangle$ together in round $r+1$. Note that this does not hinder liveness because

all non-faulty parties automatically add v to their $approvedVals$ set of round r by optimization 1 and Lemma 58.

We now prove that the $EVBCA_{Byz}$ protocol implements `EVBCA`.

Theorem 57. *The $EVBCA_{Byz}$ protocol implements `EVBCA`.*

Proof. The theorem follows from Lemmas 59, 60, 61, and 64. □

Before proving that each property of `EVBCA` is satisfied we start with a useful lemma.

Lemma 58. *If non-faulty party p_i adds v to its $approvedVals$ set of round r, then eventually v is in the round r $approvedVals$ set of all non-faulty parties.*

Proof. We use a proof by induction. Consider the $EVBCA_{Byz}$ in round $r = 0$ of the $AA_{\frac{1}{2}}$-$EVBCA_{Byz}$ protocol. The only way for a non-faulty party to add a value v to its $approvedVals$ set in this round is if it receives at least $n - t$ $\langle$echo, r, $v\rangle$ messages. At least $t + 1$ of these messages must have been sent by non-faulty parties and are therefore received by all parties. As a result, all non-faulty parties send $\langle$echo, r, $v\rangle$ messages, and all non-faulty parties add v to their $approvedVals$ set.

Now, assume the lemma is true for round $r'' > 0$, consider round $r' = r'' + 1$, and let c be the value of the common coin in round r''. If a non-faulty party adds a value to its $approvedVals$ set as a result of receiving echo messages for it from $n - t$ parties in round r', then by the same argument used for round 0, the value is eventually added to the round r' $approvedVals$ sets of all non-faulty parties. Otherwise, if a party adds c to its round r' $approvedVals$, this was because it had c in its round r'' $approvedVals$ set. Since the common coin is strong and by the assumption for round r'', all non-faulty parties add c to their round r' $approvedVals$ sets. □

Lemma 59. *The $EVBCA_{Byz}$ protocol satisfies agreement.*

Proof. This follows from Lemma 26. □

Lemma 60. *The $EVBCA_{Byz}$ protocol satisfies external validity.*

Proof. Consider a round r in which v is the only externally valid value. This means that all non-faulty parties start the round with value v and $1 - v$ was not the value of the coin in the previous round, and/or it is never in their round $r - 1$ $approvedVals$ sets. Then the only way that a non-faulty party can add $1 - v$ to their round r $approvedVals$ is if they receive echo messages for $1 - v$ in round r from at least $n - t$ parties. This cannot happen since no non-faulty party starts round r with $1 - v$. As a result, v is the only value that can be decided by non-faulty parties in round r. The rest of the proof follows from Lemma 61. □

Lemma 61. *The $EVBCA_{Byz}$ protocol satisfies termination.*

Proof. There are two possible cases to consider for a given round r of the protocol:

1. There is a single externally valid value v in round r.
2. There are two externally valid values in round r.

In case 1, if v was not the value of the coin in the previous round (meaning all non-faulty parties start round r with v and therefore send $\langle \text{echo}, r, v \rangle$ messages), termination follows from Lemma 29.

In case 1, if v was the value of the coin in the previous round, all non-faulty parties add v to their $approvedVals$ sets and directly send an echo2 message for v. All non-faulty parties then receive a sufficient number of echo2 messages for v to send echo3 messages for v. Finally, all non-faulty parties decide v.

In case 2, first we prove that all non-faulty parties send echo2 messages. If the value of the coin, c, in round $r - 1$ is added to the $approvedVals$ set of all non-faulty parties in round $r - 1$, then eventually all non-faulty parties add c to their round r $approvedVals$ set and send echo2 messages for c in round r. If this is not the case, there must be some value such that at least $t + 1$ non-faulty parties start round r with that value, and all non-faulty parties eventually receive a sufficient number of echo messages for it to send an echo2 message. Then termination follows from 29. In either case, since a non-faulty party adds a value to their $approvedVals$ prior to sending an echo2 message for it, by Lemma 58, termination follows.

□

Before proving that $EVBCA_{Byz}$ satisfies binding, we prove that there is at most one non-$\perp$ value v for which non-faulty parties send echo3 messages in a given round.

Lemma 62. *If a non-faulty party sends an echo3 message for value v in round r, then no non-faulty party sends an echo3 message for $1 - v$ in round r.*

Lemma 63. *We use a proof by contradiction. Let p_i and p_j be non-faulty parties that send echo3 messages for v_i and v_j respectively in round r. If neither party sent an early echo3 message as a result of deciding the value of the coin of round $r - 1$ in the `EVBCA` of round $r - 1$ (optimization 4), then the lemma follows from Lemma 30.*

Assume p_i sent an echo3 message for v_i as a result of optimization 4. We will show that it must be the case that $v_j = v_i$. By agreement (Lemma 59), all non-faulty parties decide v_j or $\perp$ from the $EVBCA_{Byz}$ of round $r - 1$. Therefore, since the common coin is strong, all non-faulty parties start round r with $est = v_i$. By the definition of an externally valid value, $1 - v_i$ is not externally valid. If $v_j = 1 - v_i$, p_j must have added $1 - v_i$ to its $approvedVals$ set of round r. Since $1 - v_i$ is not value of the common coin of round $r - 1$ and no non-faulty party starts round r with $est = 1 - v_i$, $v_j = v_i$.

Lemma 64. *The $EVBCA_{Byz}$ protocol satisfies binding.*

Proof. This follows from Lemma 62 and Lemma 31. □

Lemma 65. *The $ABA_{\frac{1}{2}}$-$EVBCA_{Byz}$ protocol solves asynchronous Byzantine agreement in expected 13 broadcasts in a system with $n \geqslant 3t + 1$ parties and a degree $2t$ strong coin against an adaptive adversary if `CommonCoin` is a correct implementation of a strong common coin.*

Proof. In expectation it takes 2 rounds of the $AA_{\frac{1}{2}}$-$EVBCA_{Byz}$ protocol until the coin value is equal to the non-$\perp$ decision value of non-faulty parties. Once all non-faulty parties have adopted the same value, it takes an additional two rounds in expectation for the coin to be equal that value again so they may all decide. As a result, the $AA_{\frac{1}{2}}$-$EVBCA_{Byz}$ protocol takes in expectation 4 rounds for all non-faulty parties to decide. Next, we must count the number of broadcasts that occur in each of the 4 rounds.

In the $AA_{\frac{1}{2}}$-$EVBCA_{Byz}$ protocol, in every round after the first round, at most one value, the value opposite the coin value of the previous round, is echoed by non-faulty parties. If all non-faulty parties start a round with the same value and this value is the same as the value of the coin of the previous round, non-faulty parties don't send any echo messages. There are two rounds in which non-faulty parties don't start with the same value and two rounds that they do start with the same value in expectation. As a result, in expectation there are 4 broadcasts in the first round, 2 broadcasts in the round that occurs after all non-faulty parties adopt the coin value, 3 broadcasts in the remaining two rounds, and a final broadcast to ensure that all non-faulty parties terminate totalling 13 broadcasts.

Note that since the coin is of degree $2t$, and binding is achieved once the first $t+1$ non-faulty parties send echo3 messages, the reveal coin messages can be sent with the echo3 messages so that they don't require additional broadcasts. □

2.7.2 Implementing Externally Valid Binding Crusader Agreement with Threshold Signatures

In section 2.6, we presented $BCA_{Byz,TSig}$, a protocol for binding crusader agreement that, when plugged into $AA_{\frac{1}{2}}$ to get $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$, implements Asynchronous Byzantine Agreement with a strong coin and threshold signatures. As shown in Lemma 140, the $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ protocol takes an expected 13 rounds of broadcast to terminate with a $2t$ degree strong coin. In this section, we describe two optimizations that can be applied to $BCA_{Byz,TSig}$ to obtain $EVBCA_{Byz,TSig}$ so that $AA_{\frac{1}{2}}$-$EVBCA_{Byz,TSig}$ requires 9 rounds of broadcast to terminate in expectation when used with a degree $2t$ strong coin.

We first define an externally valid value in the context of the $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ protocol and show that it satisfies Definiton 52.

Definition 66. *(Externally Valid Value for $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$) A value v is **externally valid** in round r of the $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ protocol if any of the following conditions are satisfied:*

1. *A non-faulty party starts round r with value v.*
2. *There is a $\sigma_{echo3,r',v}$ from round $r' \leqslant r$ where v was the value of the common coin in round r'.*
3. *There is a $\sigma_{echo3,r-1,v}$ from round $r-1$ where v was not the value of the common coin in round*

$r-1$.

Lemma 67. *Externally Valid Value for $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ satisfies the definition of Externally Valid Value for $AA_{\frac{1}{2}}$ in Definition 52.*

Proof. Item 1 trivially satisfies Definition 52.

Consider item 2 of Definition 66. If such a $\sigma_{\text{echo3},r',v}$ exists, by quorum intersection, all non-faulty parties must have decided $\perp$ or v from the $BCA_{Byz,TSig}$ of that round and set $est = v$. All non-faulty parties start the next round of $AA_{\frac{1}{2}}$ $BCA_{Byz,TSig}$ with v and by validity of $BCA_{Byz,TSig}$, all non-faulty parties start every round $r'' > r'$ (that they start) with v. Hence this condition satisfies Definition 66.

Finally, consider item 3 of Definition 66. If such a $\sigma_{\text{echo3},r-1,v}$ exists, then by quorum intersection, no non-faulty party could have output $1-v$ from the $BCA_{Byz,TSig}$ of that round, and no non-faulty party could have committed $1-v$. Some non-faulty must have started round $r-1$ with v, so it must have been externally valid in round $r-1$. The lemma therefore follows. □

The following two optimizations, when applied to $BCA_{Byz,TSig}$, result in $EVBCA_{Byz,TSig}$.

1. If party p_i outputs v from the $BCA_{Byz,TSig}$ in round r of $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ and therefore sets $est = val$ in round r, where val is not equal to the value of the coin (line 7 of Algorithm 2.1), then they directly send a message $\langle$echo2, est, $\sigma_{\text{echo3},r,val}\rangle$ in round $r+1$, where $\sigma_{\text{echo3},r,val}$ is the output of line 30 in Algorithm 2. In round $r+1$, other parties verify this echo2 message prior to adding it to their $pendingEcho2$ set by verifying that threshold-verify((echo3, r, val), $\sigma_{\text{echo3},r,val}$) = $true$. Upon receiving such an echo2 message, they directly send an echo2 message for val with the same $\sigma_{\text{echo3},r,val}$ in round $r+1$ if they haven't already sent an echo2 message in that round.
2. If party p_i decides v in the round r instance of $BCA_{Byz,TSig}$ and v is the value of the coin in round r of $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$, it directly sends a designated message to indicate that it is safe for all non-faulty parties to decide v that contains the $\sigma_{\text{echo3},r,v}$ created on Line 30 to all parties. Upon receiving this message and learning the value of the

coin in round r, each non-faulty party immediately decides v and forward the message to all other non-faulty parties.

Theorem 68. *The $EVBCA_{Byz,TSig}$ protocol implements `EVBCA`.*

Proof. This follows from Lemmas 70, 71, 72, and 73. □

Before proving that each of the properties of `EVBCA` are satisfied, we prove a useful lemma:

Lemma 69. *If $\sigma_{echo3,\ r,\ v}$ exists from round r and value of the common coin of round r is v, and no $\sigma_{echo3,\ r'',\ 1\text{-}v}$ exists from a round $r'' < r$ with $1 - v$ was the value of the coin in round r'', then in all rounds $r' \geqslant r$, $1 - v$ is not externally valid.*

Proof. If $\sigma_{\text{echo3, r, v}}$ exists from round r and value of the common coin of round r is v, by quorum intersection on the echo3 messages, all non-faulty parties set $est = v$ in round r. Neither of the optimizations apply for value $1 - v$, and all non-faulty parties start round $r + 1$ with value v. By validity, all non-faulty parties start every round $r' > r$, with v and σ_{echo3} for $1 - v$ is never created, thus proving the lemma.

□

We now prove that $EVBCA_{Byz,TSig}$ satisfies all the properties of `EVBCA`.

Lemma 70. *The $EVBCA_{Byz,TSig}$ protocol satisfies agreement.*

Proof. We consider two cases:

1. A non-faulty party decides v in round r upon receiving $2t + 1$ echo3 messages for v.
2. A non-faulty party decides a value v in round r upon receiving a proof $\sigma_{\text{echo3},r',v}$ from round $r' \leqslant r$ in which v was the value of the common coin.

In the first case, it must be the case that v is externally valid in round r, so by Lemma 69 $\sigma_{\text{echo3},r',1-v}$ cannot exist from a round $r' < r$ where $1 - v$ was the value of the coin in round r'. As a result, by quorum intersection, the only value a non-faulty party can decide in the $EVBCA_{Byz,TSig}$ of round r is v or $\perp$.

For the second case, by Lemma 69, a non-faulty party cannot decide $1 - v$ in round r. □

Lemma 71. *The $EVBCA_{Byz,TSig}$ protocol satisfies external validity.*

Proof. Assume that v is externally valid and $1-v$ is not externally valid. Then it must be the case all non-faulty parties start round r of the protocol with v and there is no $\sigma_{\text{echo3},r',1-v}$ from a round $r' < r$ such that threshold-verify$((\text{echo3}, r', 1-v), \sigma_{\text{echo3},r',1-v})$=*true*, where $r' = r-1$ or $r' < r$ and v was the value of the common coin in round r'. Since no non-faulty party sends an echo message in round r with $1-v$, $\sigma_{\text{echo},r,1-v}$ is never created and no non-faulty party adds $(1-v, *)$ to their *pendingEcho*2 set. Therefore, no non-faulty party ever sends an echo3 message containing $1-v$ or $\perp$ in round r, and no non-faulty party can decide $1-v$. The rest of the proof follows from that of Lemma 72.

□

Lemma 72. *The $EVBCA_{Byz,TSig}$ protocol satisfies termination.*

Proof. Consider a round r of the protocol. There are 3 cases under which we must prove termination:

1. There is no $\sigma_{\text{echo3},r,v}$ from round $r-1$ or from round $r' < r$ where v was the value of the coin.
2. A party starts round r with $\sigma_{\text{echo3},r-1,v}$ from round $r-1$ where v was not the value of the coin.
3. A party starts round r with a $\sigma_{echo,r',v}$ from round $r' < r$ where the value of the coin was v.

Termination for case 1 follows from Lemma 137.

In case 2, the party that has the $\sigma_{echo3,r-1,v}$ directly sends in round r $\langle$echo2, est, $\sigma_{\text{echo3},r,val}\rangle$, where $\sigma_{\text{echo3},r,val}$ is the output of line 30 from round $r-1$ of the protocol. Note that this party could be Byzantine and therefore send this to only some of the non-faulty parties. Every non-faulty party that receives this echo2 message and sends $\langle$echo2, est, $\sigma_{\text{echo3},r-1,v}\rangle$ if they haven't already sent an echo2 message. It is left to prove that all non-faulty parties that did not receive this message send echo2 messages. This follows from the fact that there must be some value such that at least $t+1$ non-faulty parties started with this value. As a result, every non-faulty party's *pendingEcho*2 size becomes of size at least $2t+1$ and each

non-faulty party sends an echo3 message that satisfies the conditions on Lines 20 and 23. Every non-faulty party eventually has a *pendingEcho*3 set of size $2t + 1$ and decides a value.

In case 3, some party has a proof, $\sigma_{\text{echo3},r',v}$ from round $r' < r$ in which the common coin equalled v, and therefore directly sends a designated message containing $\sigma_{\text{echo3},r',v}$ to other parties. All non-faulty parties that receive this message decide v upon receiving this message and learning the value of the coin in round r'; in addition, they forward this message to all the other parties, so all non-faulty parties decide. □

Lemma 73. *The $EVBCA_{Byz,TSig}$ protocol satisfies binding.*

Proof. For the purposes of this proof, we refer to a value v as "permanently externally valid" if $\sigma_{echo3,r',v}$ exists from a round r' where v was the value of the common coin. A non-faulty party decides a value v in round r either after receiving proof that v is permanently externally valid or after receiving echo3 messages from $2t + 1$ parties in round r.

We therefore consider two cases for a round r:

1. There is a permanently externally valid value v with a proof from a round $r' < r$.
2. There is no permanently externally valid value v from a round $r' < r$.

By Definition 66, $1 - v$ cannot be decided by a non-faulty party in round r, since that would make $1 - v$ externally valid, violating Lemma 69.

In case 2, a non-faulty party decides only after receiving echo3 messages from $2t + 1$ parties. Binding follows from Lemma 139. □

Note that it is possible that a Byzantine party has a proof $\sigma_{echo3,r,v}$ from round r in which v was the value of the coin, and no non-faulty party has the proof. In this case, all non-faulty parties would have output v or $\perp$ from $EVBCA_{Byz,TSig}$ of round r, and therefore would adopt v as their value. If the Byzantine party ever sends $\sigma_{echo3,r,v}$ to a non-faulty party, validity ensures that it will be safe for that non-faulty party to decide v and forward the proof to all of the other parties.

We now prove two lemmas regarding the expected constant round termination of the $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ protocol.

Lemma 74. *If all non-faulty parties start round r of the $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ protocol with value v, and $1 - v$ is not externally valid, then all non-faulty parties commit in a constant number of rounds.*

Proof. By external validity, all non-faulty parties output v and $\sigma_{echo3,r,v}$ is created. This happens in every round and non-faulty parties terminate in the round in which the coin equals v. In expectation, this happens in 2 rounds. □

Lemma 75. *The $AA_{\frac{1}{2}}$-$EVBCA_{Byz,TSig}$ protocol with a $2t$ degree coin takes, in expectation, 9 rounds of broadcast until all non-faulty parties terminate against an adaptive adversary.*

Proof. Note that if in any round r, there is an echo3 proof, $\sigma_{echo3,r,v}$, created, and the coin is equal to v, or there is no $\sigma_{echo3,r,v}$ created, Lemma 74 applies in the next round. It takes, in expectation, 2 rounds for the coin value to be equal to the value in the echo3 proof, at which point the non-faulty party that knows of echo3 proof commits. If the adversary chooses to have no such echo3 proof (or not let the non-faulty parties be aware of it), then all non-faulty parties output $\perp$, adopt the coin value, and commit in an expected two additional rounds. There are at most 3 broadcasts in any round of the $AA_{\frac{1}{2}}$-$EVBCA_{Byz,TSig}$ protocol. Once all non-faulty parties start a round with the same value and the echo3 proof is created in that round, there are at most 2 broadcasts per round in subsequent rounds. Therefore, it takes in expectation at most 3+3+2 broadcasts until the first non-faulty party commits and an additional 1 broadcast to ensure all non-faulty parties commit, for a total of 9 broadcasts in expectation. □

3. On the Round Complexity of Asynchronous Crusader Agreement

This work was done in collaboration with Ittai Abraham (Intel), Naama Ben-David (The Technion), and Gilad Stern (The Hebrew University of Jerusalem), with Sravya Yandamuri as lead author.

Agreement problems are at the core of many distributed systems, finding applications in replicated and reliable systems, transactional systems, cryptocurrencies, and more. It is therefore not surprising that they have gained a lot of attention in the research community, with tens of papers written about agreement problems each year. A key metric of the performance of many distributed tasks, agreement problems included, is their *round complexity*, or, intuitively, the number of sequential network round trips required to solve the task. In practice, round complexity often translates directly to latency, since communication over distributed networks is slow and forms a major bottleneck in many systems [3, 4, 30, 51, 54, 76, 81, 84, 88].

Arguably the most important and well-known agreement problem, called *consensus*, requires all non-faulty parties to unanimously agree on the same valid input value. Unfortunately, a seminal result of Fischer, Lynch and Paterson shows that no consensus algorithm can guarantee termination in an asynchronous failure-prone system [43]. Interestingly, however, weaker agreement problem variants *can* be solved in such systems, and can be sufficient for many applications.

In one such problem, known as Crusader Agreement, all parties receive an input, and non-faulty parties must output either one of the input values or a special value $\perp$. All non-faulty parties outputting a non-$\perp$ value must agree, and are only allowed to output $\perp$ if there were at least two unique input values among the non-faulty parties [31]. This weakening of consensus can be quite powerful; intuitively, if a non-$\perp$ decision represents an action, it ensures that no conflicting actions will be taken by non-faulty parties. Furthermore, CA and its variants have been used as subroutines to solve consensus in randomized protocols [1, 9, 14, 21, 70].

Our contributions

In this paper, we focus on the *Crusader Agreement (CA)* problem, and present an in-depth study of the achievable round-complexity of the problem and its variants. In particular, we consider classic CA, as well as two important variants: *Binding* Crusader Agreement (BCA) and *Graded* (Binding) Crusader Agreement (G(B)CA). In BCA, crusader agreement must be solved, but with the additional requirement that at the time at which the first non-faulty process decides its output, there exists a non-$\perp$ value v such that no non-faulty party can output a different non-$\perp$ value in any continuation of the execution. Intuitively, the adversary is *bound* to one non-$\perp$ output value and cannot adaptively affect outputs based on future knowledge. This property has recently been shown to be crucial for solving randomized consensus an an asynchronous setting [1]. In GBCA, in addition to binding, *confidence levels* or *grades* are introduced, so that parties outputting a non-$\perp$ value do so with a *grade 1* or *grade 2* label, with the guarantee that if any non-faulty party outputs v with grade 2, no non-faulty party outputs $\perp$. This variant of CA is also useful in solving randomized consensus [1]. For all of these problems, we present lower and upper bounds on their round complexity in the asynchronous model, considering both crash and Byzantine failures. We consider networks with n parties and f faulty parties.

The lower bounds for crash-resilient protocols specifically deal with protocols in which the adversary can adaptively choose the inputs of some of the parties when it schedules their first actions. While this notion of adaptive inputs might seem unnatural, when using binding crusader agreement protocols to construct consensus protocols, it is advantageous to use protocols that are also secure when the adversary is able to choose inputs adaptively, both in terms of efficiency and simplicity. For further discussion on this topic, we refer the reader to Appendix A.

We first show that binding crusader agreement (BCA) requires 2 rounds if f parties can crash and $2f + 1 \leqslant n \leqslant 3f$ in the adaptive input setting.

Theorem 76. *It is impossible to solve crash fault tolerant BCA in* 1 *round when* $2f + 1 \leqslant n \leqslant 3f$*, and the adversary can adaptively choose the inputs of the parties.*

We next turn to more complex lower bounds showing tasks where at least 3 rounds are required. First, we show that at least 3 rounds are required for crash-fault resilient graded binding crusader agreement (GBCA) if $2f + 1 \leqslant n \leqslant 3f$ in the adaptive input setting.

Theorem 77. *It is impossible to solve crash fault tolerant GBCA in* 2 *rounds when* $2f + 1 \leqslant n \leqslant 3f$*, and the adversary can adaptively choose the inputs of the parties.*

Protocols solving crash-fault tolerant BCA in 2 rounds and crash-fault tolerant GBCA in 3 rounds have been constructed in [1], showing that these lower bounds are tight.

Next, we show that at least 3 rounds are required for solving Byzantine-fault tolerant crusader agreement (CA) if there is no PKI setup and $3f + 1 \leqslant n \leqslant 4f$.

Theorem 78. *It is impossible to solve Byzantine fault tolerant CA in* 2 *rounds when* $3f + 1 \leqslant n \leqslant 4f$ *without PKI.*

We also show that this lower bound is tight in Theorem 89. Lastly, we show that the same bound holds for Byzantine-fault tolerant binding crusader agreement (BCA) if there is a PKI setup and $f \geqslant 2$, $3f + 1 \leqslant n \leqslant 4f$.

Theorem 79. *It is impossible to solve Byzantine fault tolerant BCA in 2 rounds with PKI when* $3f + 1 \leqslant n \leqslant 4f$ *and* $f \geqslant 2$*.*

The lower bounds are first proven for one (or two) failures and then generalized to an arbitrary number of failures. Somewhat surprisingly, for our lower bounds that start with $f = 2$, the generalization to arbitrary $f > 2$ requires a non-trivial argument, requiring both a stronger lower bound for the $f = 2$ case and a more intricate method of generalization (see Section 3.4).

Our Contributions: Upper Bounds

While thinking through the aforementioned lower bounds, some bounds seemed elusive and quite hard to achieve. This led us to the discovery of some surprising upper bounds. For example, the final lower bound described in the previous section looks suspiciously different from the other bounds: it only holds when $f \geqslant 2$. It turns out that the reason a more general

lower bound couldn't be constructed is that there exists a protocol solving Byzantine-fault tolerant binding crusader agreement in 2 rounds if there is a PKI setup and $n = 4, f = 1$! Following this discovery, we constructed two more protocols that work for a small number of parties but don't seem to obviously generalize to any n and f. More precisely, we construct protocols solving Byzantine-fault tolerant binding crusader agreement in 3 rounds without a PKI setup for $n = 4, f = 1$ and for $n = 7, f = 2$. The resulting protocol is also a 3-round Byzantine-fault tolerant crusader agreement protocol for any n, providing a matching upper bound to one of above lower bounds.

A key insight to constructing these protocols is to design them to be as *patient* and *conservative* as possible. By *conservative*, we mean that parties output a non-$\perp$ value only if they have to. More concretely, they output the value v only if they see that their view could have been generated in a run in which all nonfaulty parties had the input v. In this case, parties must output v; otherwise, they may violate the validity of the protocol in some run. In all other cases, parties output $\perp$. By *patient*, we mean that parties wait and output a value only when they absolutely have to. More precisely, we aim to have parties output a value only when their view could have been generated in a run of the protocol in which they may not receive any more messages. Clearly, if they do not output a value at that point, there is a run in which they never output a value. This allows us to gather as much information as possible before parties output some value.

A somewhat surprising realization is that many protocols aren't as patient as they are allowed to be. For example, many protocols simply wait to hear $n - f$ messages in a given round before proceeding to the next. On the other hand, patient protocols could wait for even more information. For example, in the second round of the protocol, parties could wait to hear both round 1 and round 2 messages from the *same* $n - f$ parties, and for each others' reports to be consistent. From our upper bounds it seems like these conditions can be quite intricate and potentially very expensive to compute for large values of n. As such, we don't suggest these protocols as realistic upper bounds, but rather almost as an impossibility result, showing that a lower bound cannot be constructed for these cases. In further work, we hope to either show that these upper bounds are general, or that a lower

bound can be constructed for some $f \geqslant 3$.

Related Work

It is well known that there are many impossibility results and lower bounds on distributed protocols [62]. Early results in the field show lower bounds on the round complexity in synchronous networks. For example, Fischer and Lynch show that $f + 1$ rounds are needed to reach Byzantine consensus in [42]. This lower bound was later generalized to authenticated protocols in [29] and [34]. In addition, similar lower bounds have been shown for synchronous crash-resilient consensus [5, 39]. Bounds are also known on early-stopping consensus, showing that at if the number of actually faulty parties is smaller than the corruption threshold, the number of needed rounds is at least 2 more than the number of corrupted parties [33].

On the other hand, fewer lower bounds are known on the round complexity of asynchronous protocols. The FLP result [43] shows that no deterministic consensus algorithm exists in an asynchronous system, even in the face of a single crash failure. More precisely, the proof shows that any consensus protocol in this setting has an infinite execution, essentially showing that the round complexity of such protocols is infinite. Similarly, the CAP theorem states that no distributed database can have consistency, availability and resilience to network partitioning [49, 63].

3.1 Model & Definitions

3.1.1 Model

* Network This work deals with a network of n parties connected via point-to-point communication channels. The network is asynchronous, meaning that there is no bound on message delay, but every message is eventually delivered in finite time. We assume that the point-to-point channels deliver messages in a FIFO order. The means that if a party sends a message m and then a message m' to the same party, the messages are delivered in that order. This can be enforced by simply adding a counter to each message, signifying when it was sent.

We model message delivery as being controlled by an adversary that can choose any delivery schedule as long as all messages are eventually delivered. We consider two types of

faults in this work: crash and Byzantine faults. In networks with crash faults the adversary may cause up to f parties to crash, meaning that those parties do not take any further actions (including receiving or sending messages). On the other hand, in networks with Byzantine faults the adversary can control up to f parties and cause them to deviate arbitrarily from the protocol.

Finally, when we say that a network has a PKI setup, we mean that each party has a well-known public key and a private key that allow it to sign messages. Every party can use the public key to check that a message was indeed sent by a given party. In addition, parties can forward received messages with their signatures, proving that the message was indeed sent by the signing party.

* Asynchronous Rounds In the synchronous setting, rounds are very clearly defined using the bound Δ on message delivery. Defining the notion of round complexity for asynchronous protocols is less straightforward [17, 56, 55], and we follow [55]. We use the idea of "causal chains" in our definition of asynchronous round complexity. Intuitively, we can think of chains of messages, with each message being sent as a result of receiving previous messages. When a message is sent, it lengthens its chain by 1, and it is considered a round k message if its chain is of length k. When mapping this behaviour to synchronous systems, all of the messages that are sent without receiving any message will be sent in round 1. Round 2 messages will be sent after receiving round 1 messages, etc.

More precisely, if a message is sent in the beginning of the protocol without receiving any other message, we consider it to be a round 1 message. If a message is sent by a nonfaulty party as a result of receiving all messages in a set M, we consider it a round $k+1$ message, where k is the maximal round number for nonfaulty messages in M (or $k = 0$ if there is no such message). We say that a party is in round k if it sent or received at least one round k message, and did not send or receive any higher-round message.

Using this notion of round complexity, we can define a k-round protocol:

Definition 80 (k-Round Protocol)**.** *A protocol is a k-round protocol if all honest parties decide a value after at most k rounds.*

Note that it is possible that protocols never terminate or do not have a bound k on the number of rounds. If this happens, these protocols can be defined as having infinite round complexity, but we deal only with finite round complexity protocols in this work.

* Adaptive Inputs We say that an adversary can choose inputs adaptively if parties only have their inputs defined by the adversary at the moment they start participating in the protocol. When dealing with binding protocols, to be defined below, this means that the binding values can only depend on the state of the nonfaulty parties that started participating in the protocol at that time, and cannot depend on the inputs of parties that haven't started participating in the protocol.

3.1.2 Definitions

We start by defining the different tasks for which we have constructed lower and upper bounds. In this work we only consider protocols in which parties decide on values but continue sending messages even after their decision. This is a very common technique in the design of asynchronous protocols, allowing parties to help each other even after they have all the information needed to complete the protocols.

Definition 81 (Crusader Agreement (CA))**.** *In a Crusader Agreement protocol, each party has either* 0 *or* 1 *as an input, and parties decide either* 0, 1 *or* $\perp$*. A Crusader Agreement protocol has the following properties:*

> ***(Agreement)*** *If two nonfaulty parties decide values x and y, then either $x = y$ or one of the values is $\perp$.*
>
> ***(Validity)*** *If all nonfaulty parties have the same input, then this is the only possible decision for nonfaulty parties.*
>
> ***(Termination)*** *All nonfaulty parties eventually decide.*

To be able to implement CA with an optimal tolerance to crash faults, we must weaken its validity property to the following:

- **(Weak Validity)** If all parties have the same input v, then all nonfaulty parties decide v.

Definition 82 (Graded Crusader Agreement (GCA)). *In a Graded Crusader Agreement protocol, each party has either* 0 *or* 1 *as an input, and parties decide on pairs* (v, g) *such that* $v \in \{0, 1, \perp\}, g \in \{0, 1, 2\}$ *and* $v = \perp$ *if and only if* $g = 0$*. A Graded Crusader Agreement protocol has the following properties:*

> ***(Graded Agreement)*** *If two nonfaulty parties decide on the pairs* $(v, g), (v', g')$*, then* $|g - g'| \leqslant 1$ *and if* $v \neq v'$*, either* $v = \perp$ *or* $v' = \perp$*.*
>
> ***(Validity)*** *If all nonfaulty parties have the same input* v*, then all nonfaulty parties decide* $(v, 2)$*.*
>
> ***(Termination)*** *All nonfaulty parties eventually decide.*

We define crash fault tolerant CA by weakening the validity property as with the non-graded version. We are also interested in the binding versions of both of these protocols. These protocols add an additional requirement that once the first nonfaulty party completes the protocol, the decision values are "bound". In a BCA protocol this means that even if the first party decides $\perp$, at that time we know which is the only possible non-$\perp$ decision value.

Definition 83 (Binding Crusader Agreement (BCA)). *A Binding Crusader Agreement protocol has all of the properties of a Crusader Agreement protocol as well as the following property:*

> ***(Binding)*** *At the time at which the first nonfaulty party to decide decides on a value, there exists a value* $b \in \{0, 1\}$ *such that no nonfaulty party decides* $1 - b$ *in any extension of this execution.*

Note that the binding property is only interesting in the case that the nonfaulty party referred to in the definition decided $\perp$. Otherwise, it trivially follows from agreement. Like in the binding definition of crusader agreement, once the first nonfaulty party decides on a value in a graded binding crusader agreement protocol, there is only one non-$\perp$ value that can be output from the protocol (with some grade).

Definition 84 (Graded Binding Crusader Agreement (GBCA)). *A Graded Binding Crusader Agreement protocol has all of the properties of a Graded Crusader Agreement protocol as well as the following property:*

(Graded Binding) *At the time at which the first nonfaulty party to decide decides on a value, there exists a value* $b \in \{0,1\}$ *such that no nonfaulty party decides either* $(1-b,2)$ *or* $(1-b,1)$ *in any extension of the protocol.*

We define crash fault tolerant BCA and GBCA by weakening the validity property as with the non-graded version.

3.2 Lower Bounds

General Proof Approach.

Each of the presented lower bounds is proven in two steps. We start by proving a lower bound for a small number of parties, setting f to be 1 or 2. We then generalize these proofs in Section 3.4. We show that if a protocol exists for some larger values of n and f, then such a protocol exists for the n and f for which we proved the original lower bound with the same round complexity. This is done by assuming that more general protocols exist, and showing that parties can simulate these protocols in the original settings (with a smaller number of parties).

For the proof of each lower bound, we construct a series of worlds. The worlds are constructed strategically to show that a party must take a certain action because their view is indistinguishable from another world where taking a different action would violate some property. In particular, we show indistinguishability with worlds where (1) all (nonfaulty) parties start with the same value, so deciding a different value would result in a violation of validity, and (2) all nonfaulty parties have sent all possible messages, so waiting for additional messages before deciding would result in a violation of termination. We put the descriptor "nonfaulty" in parenthesis where relevant due to the difference in the validity condition for crash and Byzantine fault tolerant protocols. To give the reader a hint as to the purpose of each world in our proofs, we add certain labels to the worlds.

We now describe the labels. In an **x-validity** world, all (nonfaulty) parties have input value x. In a **false x-validity** world, the view of some (nonfaulty) party is the same as in an x-validity world, causing them to decide a non-$\perp$ value (and grade 2, where relevant) even though all (nonfaulty) parties did not have the same input values. In a **maximally patient**

world, a party receives all the messages that will be sent to them by nonfaulty parties, and therefore must decide without waiting for additional messages that depend on the actions of faulty parties. For the maximally patient label, we also indicate the party that crashes, meaning another party cannot wait for messages that depend on this party before deciding without violating termination. In a **false maximally patient** world, a nonfaulty party's view is the same as in a maximally patient world, so they decide before receiving all of the messages sent by nonfaulty parties. As previously mentioned, our proofs generally proceed by constructing a chain of worlds, where there are "validity worlds" on opposite ends, and in the middle of the chain some property (binding or agreement) is violated. We indicate when a world is **symmetric** to another previously-described world on the opposite end of the chain. We use the labels **binding violation** and **agreement violation** to indicate worlds in which the properties of binding and agreement are violated, respectively.

In addition to using labels, we separate the description of each world into two bullets. The first bullet indicates the messages sent by the parties and any message delays or specific orderings where needed. The second bullet indicates the view of one or more nonfaulty parties and the actions they take accordingly.

3.2.1 Results

For our first result, we start with a simple 1 round lower bound for crash fault tolerant BCA with adaptive inputs.

Theorem 76. *It is impossible to solve crash fault tolerant BCA in* 1 *round when* $2f + 1 \leqslant n \leqslant 3f$*, and the adversary can adaptively choose the inputs of the parties.*

We show a proof for a network of three parties: p_1, p_2, and p_3. Our ultimate goal is to build up to **World 4**, in which binding is violated. In **World 4**, a party decides while p_3 lags behind; after this, the adversary adaptively chooses the input of p_3 and forces p_3 to decide 1 or 0 after a party has already decided. In order to show why p_3 decides 1 or 0 in those executions, we show indistinguishability from **World 1** or **World 2**, where all parties start with input 1 or 0, respectively. In those worlds, p_3 must decide 1 or 0 in order to not violate validity. To show why the first-deciding party decides in **World 4** without waiting for any

messages from p_3, we show indistinguishability from **World 3**, in which p_3 crashes without sending any messages. In **World 3**, parties cannot wait for messages that are dependent on p_3 before deciding, as this would result in a violation of termination.

3 party proof. **World 1 (**$1-$**validity, maximally patient for** p_2 **crash)**:

- p_1 and p_3 are nonfaulty. p_2 crashes immediately. All parties have input 1.
- p_1 and p_3 must decide 1 after receiving each other's messages without waiting for any additional messages by validity and termination.

World 2 ($0-$**validity, maximally patient for** p_1 **crash)**:

- p_2 and p_3 are nonfaulty. p_1 crashes immediately. All parties have input 0.
- p_2 and p_3 must decide 0 after recieving each other's messages without waiting for any additional messages by validity and termination.

World 3 (maximally patient for p_3 **crash)**:

- p_1 and p_2 are nonfaulty. p_3 crashes immediately. p_1 and p_3 start with inputs 1 and 0 respectively.
- p_1 and p_2 must decide after receiving each other's messages without waiting for any additional messages by termination.

World 4 (false maximally patient, false validity, binding violation):

- p_1, p_2, and p_3 are nonfaulty. p_1 starts with input 1 and p_2 starts with input 0; p_3 lags behind, and its input will be adaptively chosen later. p_1 and p_2's messages are delivered to each other, so they decide due to indistinguishability from **World 3**. The adversary now chooses one of the following extensions:
  1. p_3 has input value 1. p_1's messages are delivered to p_3, and p_2's messages are only delivered after p_3 decides.
  2. p_3 has input value 0. p_2's messages are delivered to p_3, and p_1's messages are only delivered after p_3 decides.
- In extension 1, p_3 outputs 1 due to indistinguishability from **World 1**; or in extension 2, p_3 outputs 0 due to indistiguishability from **World 0**. This constitutes a binding

violation, as we show that both 1 or 0 are possible values that p_3 decides after another party has already decided. Note that this does not imply a violation of agreement, as it is possible for the party (or parties) deciding before p_3 to decide $\perp$.

□

We now present our second result in the crash case: a 2 round lower bound for GBCA.

Theorem 77. *It is impossible to solve crash fault tolerant GBCA in* 2 *rounds when* $2f + 1 \leqslant n \leqslant 3f$*, and the adversary can adaptively choose the inputs of the parties.*

We show a proof using a network of three parties: p_1, p_2, and p_3. Our approach is to build up to a world, **World 3**, in which there is a violation of binding. The strategy of the adversary to violate binding is as follows. First, p_1 is forced to output before p_3's input value is chosen. Then, the adversary chooses p_3's input and forces them to decide 1 or 0, thus breaking binding. To show how the adversary has p_3 decide 1 or 0 in **World 3**, we present 2 symmetric sets of 3 worlds. Each set consists of the following three types of worlds:

1. A validity world showing why a party must decide a non-$\perp$ value with grade 2
2. A world where one of the parties crashes
3. A world that is both indistinguishable from the first type of world for some party other than p_3 (meaning that it decides a non-$\perp$ value with grade 2) and indistinguishable from the second type of world for p_3, showing why p_3 decides the non-$\perp$ value that it does (so as not to violate graded agreement) in each extension of **World 3** without waiting for more messages (so as not to violate termination).

For ease of exposition, we include only the worlds described in point 3 above (**World 1** and **World 2**) in the main proof of this theorem. We separate the indistinguishability arguments and the corresponding worlds into two lemmas: Lemma 85 and 86. Apart from the 2 sets of 3 symmetric worlds described above, and **World 3** in which binding is broken, we construct an additional world to show why p_1 decides in **World 3** while p_3 lags behind. This world and the corresponding indistinguishability argument are proven separately in Lemma 87. We provide the proofs for these lemmas after the proof of Theorem 77.

3 party proof. In the description of the following worlds, we only describe the runs until a specific point, and have some arbitrary message scheduling following that.

World 1 (false 1-validity, false maximally patient):

- p_1, p_2, and p_3 are nonfaulty. p_1 and p_3 have input 1, while p_2 has input 0. Initially, p_1's round 1 messages are delivered to p_2 and p_3, and then p_3's round 1 messages are delivered to p_1 and p_2. Following that, any round 2 messages that p_1 sends are delivered to p_2, and any of p_3's round 2 messages are delivered to p_1 and p_2. From this point on, p_2 and p_3's messages are delivered to each other without delay.
- By Lemma 85, p_3 decides without waiting for additional messages, and its output is of the form $(1, g)$ such that $g \in \{1, 2\}$.

World 2 (false 0-validity, false maximally patient, symmetric to World 1):

- p_1, p_2 and p_3 are nonfaulty. p_1 has input 1, and p_2 and p_3 have input 0. Initially, p_2's round 1 messages are delivered to p_1 and p_3, and then p_3's round 1 messages are delivered to p_1 and p_2. Following that, any round 2 messages that p_2 sends are delivered to p_1, and any of p_3's round 2 are delivered to p_1 and p_2. From this point on, p_1 and p_3's messages are delivered to each other without delay.
- By Lemma 86, p_3 must decide $(0, g)$ for $g \in \{1, 2\}$.

.

World 3 (binding violation, false maximally patient):

- p_1, p_2 and p_3 are nonfaulty. p_1 has input 1, p_2 has input 0, and p_3's input will be adaptively chosen by the adversary based on the value it wants p_3 to output after the first party to output does so. At the start of the execution, p_1 and p_2's round 1 messages are delivered to each other, and then any resulting round 2 messages are delivered to each other. By Lemma 87, p_1 outputs without waiting for any messages that depend on p_3 at this time. We will now show two extensions of this run, one in which p_3 outputs $(1, g)$ for some $g \in \{1, 2\}$, and one in which it outputs $(0, g)$ for some $g \in \{1, 2\}$, showing that the protocol is not binding.
    1. The adversary adaptively chooses input 1 for p_3. Following that, p_3 receives p_1's

round 1 messages, and then continues communicating freely with p_2 without any delays. At this point in time, p_3's view consists of round 1 messages from p_1 and p_2 and any round 2 messages from p_2 sent as a result as receiving p_1's round 1 messages and then p_3's round 1 messages. This view is identical to the one it has in **World 1**, so p_3 decides $(1, g)$ for some $g \in \{1, 2\}$.

2. The adversary adaptively chooses input 0 for p_3. Following that, p_3 receives p_2's round 1 messages, and then continues communicating freely with p_1 without any delays.

   At this point in time, p_3's view consists of round 1 messages from p_1 and p_2 and any round 2 messages from p_1 sent as a result as receiving p_2's round 1 messages and then p_3's round 1 messages. This view is identical to the one it has in **World 2**, so p_3 decides $(0, g)$ for some $g \in \{1, 2\}$.

□

Lemma 85. *In **World 1** from the proof of Theorem 77, p_3 must decide (1, g) for $g \in \{1, 2\}$ without waiting for any round 2 messages from p_1.*

Proof. **World 1.a) (1-validity, maximally patient for p_2 crash)**:

- p_1 and p_3 are nonfaulty. p_2 crashes without sending any initial messages. All three parties start with input 1. p_1 and p_3 communicate without delay.
- p_1 and p_3 must decide $(1, 2)$ without waiting for any messages from p_2 by validity and termination.

World 1.b) (maximally patient for p_1 crash):

- p_1 and p_3 have input 1, while p_2 has input 0. p_1 is faulty, sends round 1 messages, which are delivered to both p_2 and p_3, and then p_1 crashes. Following that, p_3's round 1 messages are delivered to p_2. Finally, p_2 and p_3's messages are delivered to each other without delay.
- Because p_1 crashed, p_2 and p_3 must decide without waiting for any round 2 messages sent by p_1, by termination.

We now argue why in **World 1** from the proof of Theorem 77, p_3 must decide $(1, g)$ such

that $g \in \{1, 2\}$ without waiting for any round 2 messages from p_1. First, we show that p_1 decides (1, 2), in **World 1**. Observe that p_1's view in **World 1** is indistinguishable from its view in **World 1.a** because p_1 and p_3 have input 1 and they start by exchanging both round 1 and round 2 messages. It follows that p_1 decides (1, 2), and thus when p_3 decides some value, it must decide $(1, g)$ such that $g \in \{1, 2\}$ by graded agreement. Next, we argue that p_3 must decide in **World 1** without waiting for any round 2 messages from p_1. Observe that in **World 1**, since p_1's messages (apart from any round 1 messages) are delayed for p_3, p_3's view is indistinguishable from its view in **World 1.b**. As a result, p_3 must not wait for any round 2 messages from p_1 before deciding so as not to violate termination. Note that p_2 cannot send any messages which rely on p_1's round 2 messages, because this is a 2-round protocol, so p_3's view is indeed indistinguishable in both worlds. □

Lemma 86. *In **World 2** from the proof of Theorem 77, p_3 must decide (0, g) for $g \in \{1, 2\}$ without waiting for any round 2 messages from p_2.*

Proof. **World 2.a) (0-validity, maximally patient for p_1 crash, symmetric to World 1.a)**:

- All three parties have the input 0. p_2 and p_3 are nonfaulty, and p_1 crashes prior to sending any messages.
- p_2 and p_3 must decide (0, 2) without waiting for any messages dependent on p_1 by validity and termination.

World 2.b) (maximally patient for p_2 crash, symmetric to World 1.b):

- p_1 has input 1, while p_2 and p_3 start with input 0. p_2 sends round 1 messages, which are delivered to both p_1 and p_3, and then p_2 crashes. Following that, p_3's round 1 messages are delivered to p_1. Finally, p_1 and p_3's messages are delivered to each other without delay.
- Because p_2 crashed, p_1 and p_3 must decide without waiting for any additional messages from p_2, by termination.

We now argue why p_3 must decide $(0, g)$ for $g \in \{1, 2\}$ in **World 2** without waiting for any of p_2's round 2 messages. First, we show that p_2 decides (0, 2). Since p_1's messages are initially delayed, p_2 decides $(0, 2)$ due to indistinguishability from **World 2.a**, in which

p_1 crashes. As a result, if p_3 decides, it must decide $(0, g)$ such that $g \in \{1, 2\}$ so as not to violate graded agreement. Next, we show why p_3 decides without waiting for any round 2 messages from p_2. This follows an indistinguishability argument with **World 2.b** for p_3, since any messages from p_2 apart from its round 1 messages are delayed for p_3 in **World 2**. □

Lemma 87. *In **World 3** from the proof of Theorem 77, p_1 must output without waiting for any messages that depend on p_3.*

Proof. **World 3.a) (maximally patient for p_3 crash)**:

- p_1 and p_2 are nonfaulty, while p_3 crashes immediately before sending any messages. p_1 has input 1 and p_2 has input 0.
- p_1 and p_2 must decide without waiting for any messages dependent on p_3 by termination.

The lemma follows from a straightforward indistinguishability argument from **World 3.a)**, as any messages from p_3 and dependent on p_3 are delayed for p_1 in **World 3**. □

For our third result, we show a lower bound for Byzantine fault tolerant CA without PKI. With a Byzantine adversary and no PKI, the faulty parties are able to simulate receiving certain messages from nonfaulty parties.

Theorem 78. *It is impossible to solve Byzantine fault tolerant CA in 2 rounds when $3f + 1 \leqslant n \leqslant 4f$ without PKI.*

We present a proof for 4 parties: p_1, p_2, p_3 and p_4. In this proof, we build up to **World 5** in which agreement is violated because nonfaulty parties p_1 and p_4 decide 1 and 0, respectively. We start by showing two maximally patient worlds (**World 1** and **World 2**), where one party has omission failures and sends its input value message only to one other party. By termination, the nonfaulty parties must not wait to hear more messages before deciding. We then show two symmetric validity worlds (**World 3** and **World 4**) in which a Byzantine party simulates receiving a message from a non-faulty party that it didn't send. Due to indistinguishability from the maximally patient worlds, honest parties must decide

without waiting for additional messages, but they must decide non-$\perp$ values by validity. Finally, in **World 5**, the adversary uses a Byzantine p_3 to have p_1 and p_4 decide different non-$\perp$ values using indistinguishability from the previously defined worlds.

4 party proof. In the following discussion, when we say that parties p_1, p_2 and p_3 have each other's messages delivered, we mean that the party receives its own messages first, and then p_1's messages are delivered first, then p_2's and then p_3's (similarly for p_2, p_3 and p_4).

World 1 (maximally patient for p_4 crash):

- All parties except p_4 are nonfaulty. p_4 crashes immediately without sending any messages. p_1 and p_2 have input 1; p_3 and p_4 have input 0. p_1, p_2 and p_3 have their round 1 messages delivered to each other, and then any round 2 messages that they send as a result are delivered to each other.
- All nonfaulty parties must decide without waiting for any messages dependent on p_4.

World 2 (maximally patient for p_1 omission, symmetric to World 1):

- All parties other than p_1 are nonfaulty; p_1 has omission failures. p_1 and p_2 have input 1, while p_3 and p_4 have input 0. p_1 sends round 1 messages as an honest party would with input 1 only to party p_2, and the messages are delivered first for p_2. Following that, p_2, p_3 and p_4 have their round 1 messages delivered to each other, and then any round 2 messages that they send as a result are delivered to each other.
- All nonfaulty parties must decide without waiting for any more messages from p_1 by termination.

World 3 (0-validity, false maximally patient, simulation):

- All parties except for p_2 are nonfaulty. p_2 is Byzantine. p_1, p_3 and p_4 start with 0. p_2 acts as if it started with input 1 and simulates p_1 starting with input 1. All messages from p_1 are delayed to p_3 and p_4, until they both decide. p_2 acts as if it is a nonfaulty party with input 1 such that the first message it received was a round 1 message from an honest p_1 with input 1. Following that, p_2, p_3 and p_4 have their round 1 messages delivered to each other, and then any round 2 messages that they send as a result are delivered to each other.

- Due to indistinguishability from **World 2**, p_4 decides without waiting for any additional messages. By validity, p_4 decides 0.

World 4 (1-validity, false maximally patient, simulation, symmetric to World 3):

- p_3 is Byzantine, and the remaining parties are nonfaulty. p_1, p_2, and p_4 start with input 1; p_3 acts as if it nonfaulty and has the input 0. All messages from p_4 are delayed to p_1 and p_2. p_1, p_2 and p_3 have their round 1 messages delivered to each other, and then their round 2 messages delivered to each other.
- Due to indistinguishability from **World 1**, p_1 decides before receiving any messages from p_4. By validity, p_1 decides 1.

World 5 (agreement violation, false maximally patient, false validity):

- p_3 is Byzantine, and the remaining parties are nonfaulty. p_1 and p_2 have input 1, while p_3 and p_4 have input 0. p_3 starts by acting as a nonfaulty party would with input 0. Parties p_1, p_2 and p_3's round 1 messages are delivered to each other, and then any round 2 message that they sent as a result of receiving the round 1 messages. Following that, p_3 acts as if it did not receive any round 1 messages from p_1. Now, p_4's round 1 messages are delivered to p_2 and p_3, and their round 1 messages are delivered to p_4. Finally, all round 2 messages sent by p_2 and p_3 are delivered to p_4.
- This world is indistinguishable from **World 4** for p_1 since it exchanged round 1 and round 2 messages with parties p_2 and p_3 with the same inputs without hearing from p_4. In addition, this world is indistinguishable from **World 3** for p_4 because p_1 acts as if it first received round 1 messages from p_1 with input 1, and then p_2, p_3 and p_4 exchange round 1 and round 2 messages without receiving any further messages from p_1. Therefore, p_1 and p_4 decide 1 and 0 respectively, violating the agreement property.

□

For our second lower bound in the Byzantine case, we prove the impossibility of Byzantine fault tolerant BCA with PKI in 2 rounds when $f \geqslant 2$. Since there is PKI, the faulty parties can no longer simulate receiving messages from nonfaulty parties. This necessitates a slightly more complex approach than that required for the previous lower bound.

Theorem 79. *It is impossible to solve Byzantine fault tolerant BCA in 2 rounds with PKI when* $3f + 1 \leqslant n \leqslant 4f$ *and* $f \geqslant 2$.

In this proof, we build up to a **World 6** where we show a binding violation by having an extension where a nonfaulty p_1 decides 1 and an extension where a nonfaulty p_7 decides 0 after another nonfaulty party p_5 decides. Unlike in the proof of the previous lower bound, we can no longer rely on simulation due to the presence of PKI. If we want a nonfaulty party to decide a non-$\perp$ value $v \in \{0, 1\}$, it can hear that at most $f = 2$ parties started with $1 - v$. This is because, in order to argue that a party must decide a non-$\perp$ value in a given world, we show that this party's view is indistinguishable from its view in another world in which all nonfaulty parties started with that value, enabling us to invoke validity. With PKI, if a party hears that more than f parties started with the value opposite its input value, then it knows that it is not in a validity world. As such, when attempting to understand this proof it is helpful to work backwards, starting from **World 6** to see the views of p_1 and p_7 when they decide 1 and 0, respectively. The maximally patient worlds **World 1**, **World 2**, and **World 5** show why p_1, p_5, and p_7 decide without waiting for additional messages in **World 6**. To show why the views of p_1 and p_7 are indistinguishable from validity worlds, forcing them to decide 1 and 0 respectively, we show **World 3** and **World 4** in which the honest parties all start with the same value.

Proof. As in previous proofs, when we say a party receives messages from a list of parties, they receive the messages in the listed order. For example, if a party receives messages from $p_1, \ldots, p_4$, it receives the messages from p_1 first, then p_2, and so on.

World 1 (maximally patient for p_2 and p_1 crash):

- All parties except p_1 and p_2 are nonfaulty. p_1 and p_2 crash immediately without sending any messages. p_3 and p_4 start with input 1, while p_5, p_6 and p_7 start with input 0.
- All nonfaulty parties must decide without waiting for any messages dependent on p_1 or p_2; otherwise, termination is violated.

World 2 (maximally patient for p_5 crash and p_6 omission):

- All parties except p_5 and p_6 are nonfaulty. p_1, p_2, p_3, and p_4 start with input 1. p_6 and p_7 start with input 0. p_5 crashes immediately without sending any messages. p_6 is omission failure; all messages except for any round 1 messages it sends to p_2 are omitted, and these messages are delivered for p_2 before any messages from any other parties.
- Nonfaulty parties must decide without waiting for any messages dependent on p_5 or any messages dependent on p_6 (other than any round 1 messages it sends to p_2); otherwise, termination is violated.

World 3 (0-validity, false maximally patient):

- p_3 and p_4 are Byzantine and have input 1. The rest of the parties are honest and start with input 0. All messages from p_1 and p_2 are delayed for the other parties. p_3, p_4, p_5, p_6 and p_7 exchange the same messages as in **World 1** and in the same order.
- This world is indistinguishable from **World 1** for p_7. Therefore, it decides without waiting for any additional messages. By validity, p_7 decides 0.

World 4 (1-validity, false maximally patient):

- p_6 and p_7 are Byzantine and start with input 0; the rest of the parties are honest and start with input 1. All messages from p_5 are delayed for the other parties. p_6 doesn't send any messages except for any round 1 messages that it would have sent to p_2 if it was honest, and as in **World 2**, this message is delivered for p_2 before any messages from any other parties. p_1, p_2, p_3, p_4 and p_7 send the same messages in the same order as in **World 2**.
- The world is indistinguishable from **World 2** for p_1, so it decides without waiting for any additional messages. By validity, p_1 decides 1.

World 5 (maximally patient for p_7 and p_1 omission):

- All parties except for p_1 and p_7 are nonfaulty. $p_1, \ldots, p_4$ start with input 1 and $p_5, \ldots, p_7$ start with input 0. All honest parties start by sending their round 1 messages. p_7 crashes immediately after sending its round 1 messages to all of the other parties. p_1 is omission failure, and the only message it sends is its round 1 message to p_2. p_2 receives round 1 messages from p_6 first, then from $p_1, \ldots, p_4$ and p_7, and finally from

p_5. p_2 sends round 2 messages as a result of receiving the aforementioned round 1 messages. Parties $p_3, \ldots, p_6$ receive round 1 messages from $p_3, \ldots, p_7$ and send any resulting round 2 messages. They receive any round 1 messages from p_2 following that, and possibly send additional round 2 messages. Finally, p_5 receives all round 2 messages from parties $p_2, \ldots, p_6$.

- Note that parties $p_2, \ldots, p_6$ received all round 1 messages sent by each other, and p_5 received any round 2 message sent as a result from these parties as well. This means that p_5 receives all messages from nonfaulty parties in this world, and thus by termination, p_5 decides without waiting for any additional messages.

World 6 (binding violation, false maximally patient):

- p_3 and p_4 are Byzantine, and the remaining parties are nonfaulty. $p_1, \ldots, p_4$ have the input 1 and $p_5, \ldots, p_7$ have the input 1, like **World 5**. Initially, all messages from other parties are delayed for p_7 and p_1. In addition, messages from p_1 are delayed for $p_3, \ldots, p_6$. The beginning of the run is exactly the same the run in **World 5** for $p_2, \ldots, p_6$, with p_3, p_4 sending the required messages only to parties $p_2, \ldots, p_6$ and not to p_1, p_7. Since p_5's view is identical to one which causes it to decide, it decides some value in this world as well. Next, we show the two executions in which the adversary can get p_1 to decide 1 or p_7 to decide 0, which would mean the protocol isn't binding.
  - **(Extension where p_1 decides 1)** p_1 and p_7 start by receiving round 1 messages from $p_1, \ldots, p_4, p_7$. p_1 then receives any round 2 messages from $p_1, \ldots, p_4, p_7$ except for p_2 final round 2 message sent by p_2 as a result of receiving p_5's round 1 message (which it received last). In the above, p_3 and p_4 are Byzantine, and they only send p_1 the round 2 messages they would have as a result of receiving round 1 messages from $p_1, \ldots, p_4, p_7$. Note that p_1 receives round 1 messages from $p_1, \ldots, p_4, p_7$ and then round 2 messages from $p_1, \ldots, p_4, p_7$ corresponding to p_2 receiving p_6's round 1 messages first, and then all of the parties receiving round 1 messages from each other. p_1's view is identical to the view it would have in **World 4**, so it decides 1.
  - **(Extension where p_7 decides 0)** p_7 sees round 1 messages from $p_3, \ldots, p_6$, and

then all round 2 messages that they sent as a result of receiving round 1 messages from $p_3, \ldots, p_7$. Note that they received round 1 message from p_1, p_2 only after receiving those messages. At this point, p_7's view is identical to its view in **World 3**, so it decides 0.

□

Remark 1. *It is possible to define* $S = \{p_2, p_3, p_5, p_7\}$ *and* $T = \{p_1, p_4, p_6\}$. *For these sets,* $S \cup T = \{p_1, \ldots, p_7\}$, $S \cap T = \varnothing$ *and* $|S| = 4$, $|T| = 3$. *In the proof of Theorem 79, the adversary always corrupts at most one party in S and one party in T. From Theorem 94 we can conclude that no 2-round Byzantine fault tolerant protocol exists even for any* $3f + 1 \leqslant n \leqslant 4f$ *and* $f \geqslant 2$.

3.3 Upper Bounds

Notation. The notation for a message from a party p_i is i. The initial message from a party is a special case, as it also contains a subscript $v \in \{0, 1\}$ indicating the party's input value. The first message in a valid chain of messages is always an initial message of this form. Chains of messages are separated by the operator $\cdot$. As an example, $\langle i_1 \cdot j \rangle$ is a length two chain where p_j is forwarding the initial message of p_i, where p_i has input value 1. We define the notion of a prefix of a chain recursively. Message chain C' is a prefix of chain C if $C' = C$ or there exists a party p_j such that $\langle C' \cdot j \rangle = \langle C \rangle$. We say that a message chain C *depends on* party p_i if the first message in the chain is of the form i_x such that $x \in \{0, 1\}$ or there exists a prefix of chain C, P, such that $\langle P \cdot i \rangle$ is also a prefix of chain C.

3.3.1 Results

The following upper bounds are designed such that parties forward any message they receive each other and wait for as long as they can (or nearly as much as they can). By this we mean that parties only decide on values if the messages they received could have been all messages nonfaulty parties ever send throughout an execution of the protocol. The protocols are also conservative in the sense that parties default to outputting $\perp$ unless doing so might lead to a validity violation. A party is forced to output a value $x \neq \perp$ if its view could have been obtained in an execution in which all nonfaulty parties have the input x.

The protocol described in Algorithm 3.1 is designed to work as described above. Parties start by sending their signed input to all parties, and then forwarding that input to all parties. Whenever a party receives a signed input message it forwards that message to all parties. Every party p_i then waits until there are three parties (including itself) such that p_i received all of these parties' inputs, and the messages forwarding each other's inputs. Once that happens, p_i chooses whether to output the value x that it received as input, or the value $\perp$. If p_i saw that more than one party reported its input as $1-x$ (either by receiving its input directly, or by receiving a forwarded input), p_i outputs $\perp$. Otherwise, p_i outputs x.

Input: x
1: $fwdVals_1 = fwdVals_2 = fwdVals_3 = fwdVals_4 = \{\}$, $initVals = \{\}$
2: send $\langle i_x \rangle$ and $\langle i_x \cdot i \rangle$ to all, $fwdVals_i = fwdVals_i \cup \{i_x\}$
3: **upon** receiving $\langle k_v \rangle$ from p_k and not having forwarded a message from p_k:
4: send $\langle k_v \cdot i \rangle$ to all
5: $fwdVals_i = fwdVals_i \cup \{k_v\}$
6: $initVals = initVals \cup \{k_v\}$
7: **upon** receiving $\langle j_v \cdot k \rangle$ from p_k
8: $initVals = initVals \cup \{j_v\}$
9: **if** j_{1-v} hasn't been added to $fwdVals_k$: $fwdVals_k = fwdVals_k \cup \{j_v\}$
10: **upon** $\exists p_j, p_k \neq p_i$ s.t. i_x, k_v, and $j_{v'}$ are in $fwdVals_i \cap fwdVals_k \cap fwdVals_j$ s.t. $v, v' \in \{0,1\}$:
11: let S be the set $\{s | s_{1-x} \in initVals\}$
12: **if** $|S| \leqslant 1$ **then** decide x
13: **else**, decide $\perp$

FIGURE 3.1: 4-party authenticated Asynchronous BCA for Byzantine faults for party p_i

Theorem 88. *The protocol in Figure 3.1 solves Byzantine fault tolerant BCA in 2 rounds with a PKI setup when there are* 4 *parties,* $n = 3f + 1$.

Proof. **Termination.** Termination follows from the fact that there are at least 3 honest parties, and they all will eventually receive and forward each others' initial messages.

Validity. Assume all nonfaulty parties have the same input $x \in \{0,1\}$. Parties only add values j_y to $initVals$ after receiving a message j_y, which contains j's signature on the value y. Nonfaulty parties only sign such messages with their input x, so nonfaulty parties can receive one signature on $1-x$ by the single faulty party. Therefore, if some nonfaulty party decide on some value, it will see that $|S| \leqslant 1$ in line 12 and decide x.

Agreement. Assume by way of contradiction that two parties p_i and p_j output 1 and 0 respectively. Before deciding, each of those parties waited to hear at least 3 forwarded messages from at least 3 parties. Since there are 4 parties, and at most 1 Byzantine party, they have at least one such nonfaulty party in common. Let that party be p_k. At the time p_i terminated, it heard at most one forwarded input of 0, meaning that in p_k's first 3 forwarded messages, it sent at least two messages with the value 1. Therefore, before terminating p_j heard at least two forwarded 1 inputs, and thus it could not have output 0.

Binding. Assume without loss of generality that party p_1 is the first nonfaulty party to output some value. If it outputs a value $b \neq \perp$, then we can define b to be the binding value, and the binding property trivially holds because of the agreement property. Otherwise, let I be the indices of the parties that caused p_1 to terminate, and let G be the nonfaulty parties among them. Without loss of generality, assume that $I = \{1, 2, 3\}$ and that p_2 is nonfaulty (and possibly also p_3). For each $i \in G$, define $m_{1,i}, m_{2,i}, m_{3,i}$ to be the first three values echoed by p_i, and define m_i to be the most common value among $m_{1,i}, m_{2,i}, m_{3,i}$. Now, define b to be the most common value in the multiset $\{m_i | i \in G\}$, if such a value is uniquely defined. If there is no single most common value, define b to be p_4's input, which we will show is defined by this point in time.

If b is the most common value in the multiset $\{m_i | i \in G\}$, then at least two nonfaulty parties in G sent at least two echoes with the value b in their first three echoes. Any nonfaulty party that terminates must hear at least three echoes from at least one of those parties, and thus will not output $1 - b$. Otherwise, the most common value in $\{m_i | i \in G\}$ is not uniquely defined. This must mean p_3 is faulty and thus $G = \{p_1, p_2\}$. In addition, since the most common value is not defined, $m_1 \neq m_2$. Note that p_1 and p_2 agree on the value sent by p_3, so it cannot be the case that their first three echoed values are echoes of messages sent by the same three parties. In other words, at least one of them heard from p_4, and thus p_4's input is already defined to be some value x_4. We defined $b = x_4$ as above, and all that is left to show is that no party outputs $1 - b$. We already know that p_1 output $\perp$, and by construction, p_4 cannot output $1 - b = 1 - x_4$. Therefore, only p_2 might output $1 - b$ if that was its input. If p_2's first three echoes contain the value b twice, it would not output

$1 - b$. Otherwise, $m_2 = 1 - b$ and thus $m_1 = b$. This means that if p_2 hears three echoes from p_1 before terminating, it will hear at least two echoes with the value $m_1 = b$ and will output $\perp$. Otherwise, before terminating it hears an input message from p_3 and p_4, as well as echoes of each others' values. In addition, it hears p_1's input before terminating, because p_1 is the first party to terminate and it heard p_2 echo its value at that time. In other words, p_2 hears all parties' input messages before terminating. As shown above, p_1 echoes three messages, with the input value m_1 appearing twice. Similarly, p_2 echoes three messages with the input value m_2 appearing twice. Finally, both forward the same message from p_3, and thus in total p_2 receives two messages with the value m_1 and two messages with the value m_2 before terminating. Since $m_1 \neq m_2$, in that case it outputs $\perp$. In other words, in all cases p_2 either outputs $\perp$ or b.

Round Complexity. We now prove that the protocol requires only 2 rounds. This follows from the fact that the only messages sent by honest parties are their initial messages with their input values (which don't depend on any other messages), and messages forwarding the initial messages of other parties. □

Similarly to the previous protocol, in the protocol described in Figure 3.2, parties start by sending each other their inputs. They then forward any received input and any message forwarding an input, also indicating the messages' senders. Every party p_i then waits until there are three parties (including himself) that report consistent information about each other's messages. More specifically, they forward the same messages about each other as the messages the p_i received and forwarded. Then, p_i outputs its input x if it forwarded at most one input message with the value $1 - x$ and at most one of the three aforementioned parties forwarded more than one input message with the value $1 - x$. Otherwise, p_i outputs $\perp$.

We show that this protocol is a CA protocol for any number of parties n such that $n \geqslant 3f + 1$ in Theorem 89. We then proceed to show that the protocol is also binding for $n = 4, f = 1$ and $n = 7, f \geqslant 2$ in Theorems 91 and 92 respectively, meaning that in these cases it is also a BCA protocol.

Input: x

1: $coreSet_i = \{\}$
2: **for** $j \in 1 \dots n$:
3: $\quad initVals_j = \{\}$
4: $\quad$ **for** $k \in 1 \dots n$:
5: $\quad\quad fwdedMsgs_{j,k} = []$
6: send $\langle i_x \rangle$ to all
7: **upon** receiving $\langle j_v \rangle$ from p_j and $fwdedMsgs_{i,j} = []$:
8: $\quad$ send $\langle j_v \cdot i \rangle$ to all
9: $\quad initVals_i = initVals_i \cup \{j_v\}$
10: $\quad fwdedMsgs_{i,j} = fwdedMsgs_{i,j}.append(j_v)$
11: **upon** receiving $\langle k_v \cdot j \rangle$ from p_j and $k_* \cdot j \notin fwdedMsgs_{i,j}$:
12: $\quad$ send $\langle k_v \cdot j \cdot i \rangle$ to all
13: $\quad initVals_j = initVals_j \cup \{k_v\}$
14: $\quad fwdedMsgs_{i,j} = fwdedMsgs_{i,j}.append(k_v \cdot j)$
15: $\quad fwdedMsgs_{j,k} = fwdedMsgs_{j,k}.append(k_v)$
16: **upon** receiving $\langle k_v \cdot l \cdot j \rangle$ from p_j and having received $k_v \cdot l$ from p_l:
17: $\quad fwdedMsgs_{j,l} = fwdedMsgs_{j,l}.append(k_v \cdot l)$
18: **upon** $\exists$ a set of $n - f$ distinct parties $coreSet_i$ s.t. the following 3 conditions hold:
$\quad$ 1. $p_i \in coreSet_i$
$\quad$ 2. $\forall (j,k,l) \in coreSet_i$, $fwdedMsgs_{j,k} = fwdedMsgs_{l,k}$
$\quad$ 3. $\forall j \in coreSet_i$, $\exists v \in \{0,1\}$ s.t. $fwdedMsgs_{i,j}[1] = v_j$ and $\forall k \in coreSet_i$, $v_j \in initVals_k$
19: $\quad \forall j \in \{1 \dots n\}$ let $S_j = \{s | s_{1-x} \in initVals_j\}$
20: $\quad$ **if** $|S_i| \leqslant f$ and $|\{j \in \{1 \dots n\} \text{ s.t. } |S_j| > f\}| \leqslant f$:
21: $\quad\quad$ decide x
22: $\quad$ **else** decide $\perp$

FIGURE 3.2: 7-party unauthenticated Asynchronous BCA for Byzantine faults for party p_i

Theorem 89. *The protocol in Figure 3.2 solves Byzantine fault tolerant CA for $n \geqslant 3f + 1$ parties in 3 communication rounds without a PKI setup.*

Proof. **Validity.** Assume all nonfaulty parties have the input b, and that some nonfaulty party p_i outputs some value. At that time, it received the message $\langle j_{1-b} \rangle$ from at most f parties, and thus $|S_i| \leqslant f$. In addition, every nonfaulty p_j only sends $\langle k_x \cdot j \rangle$ messages after receiving a $\langle k_x \rangle$ message from p_k. This means that each p_j sends at most f such messages with $x = 1 - b$, and thus for every nonfaulty p_j, $|S_j| \leqslant f$. Therefore, both conditions of line 20 hold, and thus p_i outputs b, as required.

Agreement. Assume by way of contradiction that two nonfaulty parties p and q output 0 and 1 respectively. Define $coreSet_0$ and $coreSet_1$ to be the sets $coreSet$ they have at the time they output their respective values. Define $coreSet_{0,1} = coreSet_0 \cap coreSet_1$, and note that $|coreSet_{0,1}| \geqslant f + 1$ because $|coreSet_0| = |coreSet_1| = n - f$. There are at most f Byzantine

parties, so let p_i be a nonfaulty party in $coreSet_{0,1}$. Both p and q completed the protocol with p_i in their respective core sets, so it saw that it forwarded the initial value messages sent by all parties in their respective cores. Assume without loss of generality that p_i sent messages of the form $\langle k_v \cdot j \cdot i \rangle$ for each pair of parties $p_j, p_k \in coreSet_0$ before it did so for all such pairs of parties in $coreSet_1$. From condition 2 of line 18, p received the messages $\langle k_v \cdot j \rangle$ from each such p_j as well as k_v from p_k. From the first condition of line 20, p_i saw that at most f of those k_v messages had $v = 1$, because otherwise $|S_i| > f$ would have been true, and p_i would have output $\perp$ instead. On the other hand, p_i forwarded the same messages in the same order to q. This means that for every $p_j \in coreSet_{0,1}$, p_i forwarded at least $f + 1$ messages of the form $\langle k_0 \cdot j \cdot i \rangle$ before forwarding the final message required for q to terminate. From the second condition of line 18, q waits to hear the messages $\langle k_0 \cdot j \rangle$ from the parties $p_j \in coreSet_{0,1}$, and thus when it terminates, it sees that at least $f + 1$ parties in $coreSet_1$ have forwarded at least $f + 1$ initial 0 values, causing it not to output 1 and reaching a contradiction.

Termination. All nonfaulty parties eventually send their input messages. After receiving those messages, every p_i sends a $\langle j_x \cdot i \rangle$ for every $\langle j_x \rangle$ message it received. Similarly, every p_i sends a $\langle k_x \cdot j \cdot i \rangle$ for every $\langle k_x \cdot j \rangle$ message it received. After receiving the all of these messages from each nonfaulty party, every nonfaulty party has the conditions of line 18 hold with respect to the $n - f$ nonfaulty parties, and thus every nonfaulty party decides some value if it hadn't done so previously.

Round Complexity. Parties send at most chains of length 3, and thus the protocol is a 3-round protocol. □

We now turn to show that the protocol is also binding in the case of $n = 4, f = 1$ and $n = 7, f \leqslant 2$.

Lemma 90. *Let p_i and p_j be two nonfaulty parties. If $p_k \in coreSet_i \cap coreSet_j$, then $\forall p_l \in coreSet_i$ and $\forall p_m \in coreSet_j$, either p_i's $fwdedMsgs_{l,k}$ is a prefix of p_j's $fwdedMsgs_{m,k}$ or p_j's $fwdedMsgs_{m,k}$ is a prefix of p_i's $fwdedMsgs_{l,k}$.*

Proof. By quorum intersection, $\left|coreSet_i \cap coreSet_j\right| \geqslant f + 1$ and at least one of the parties

in the intersection must be nonfaulty. The lemma follows from condition 2 on line 18. □

Theorem 91. *The protocol in Figure 3.2 solves Byzantine fault tolerant BCA for $n = 4$ parties and $f = 1$ in 3 communication rounds without a PKI setup.*

Proof. As shown in Theorem 89, the protocol is a 3-round CA protocol for any $n \geqslant 3f + 1$, and thus it has the Validity, Agreement and Termination properties. All that is left to show is that the protocol is also binding.

Binding. Assume that the first nonfaulty party to output outputs $\perp$ (otherwise binding follows from agreement). W.l.o.g. assume p_2 is the first nonfaulty party to output, that it started with input value 1, and consider the set $coreSet_2$ at the time that p_2 decides. For binding not to hold, there must be an extension of this execution where some nonfaulty party decides 1 and one in which a nonfaulty party decides 0. W.l.o.g. assume that p_1 is the nonfaulty party who can decide 1 and p_4 is the nonfaulty party who can decide 0. Note that parties only decide a non-$\perp$ value if that was their input value. As such, p_1 must have started with 1 and p_4 must have started with 0. If p_1, p_2 and p_4 are in $coreSet_2$, then p_4 cannot later output 0 by the condition on line 20. Since $|coreSet_2| \geqslant 3$, there are two possible cases:

1. p_1, p_2 and p_3 are in $coreSet_2$. Then p_1 and p_2 both forward the messages $\langle 1_1 \cdot 3 \rangle$ and $\langle 2_1 \cdot 3 \rangle$. Since only one of p_1 and p_2 can be in $coreSet_4$ for p_4 to output 0, p_3 must be in $coreSet_4$. Assume first that p_2 is the other party in $coreSet_4$. For p_4 to be able to decide 0, it must not hear the messages $\langle 1_1 \cdot 3 \cdot 2 \rangle$ and $\langle 2_1 \cdot 3 \cdot 2 \rangle$ from p_2, or $coreSet_4$ will not satisfy the condition on line 20 for p_4 to output 0 (since p_4 will wait to hear that p_3 forwarded the initial values of p_1 and p_2 or it will hear messages from p_1). So p_2 must send all the messages necessary for p_4 to output 0 before it sends the messages $\langle 1_1 \cdot 3 \cdot 2 \rangle$ and $\langle 2_1 \cdot 3 \cdot 2 \rangle$. This necessarily includes the messages $\langle 4_0 \cdot 3 \cdot 2 \rangle$ and $\langle 3_0 \cdot 3 \cdot 2 \rangle$, as well as $\langle 4_0 \cdot 2 \cdot 2 \rangle$ and $\langle 3_0 \cdot 2 \cdot 2 \rangle$, and $\langle 4_0 \cdot 4 \cdot 2 \rangle$ and $\langle 3_0 \cdot 4 \cdot 2 \rangle$. If this happens, it cannot be the case that $coreSet_1$ satisfies the conditions for p_1 to output 1, and we have arrived at a contradiction. If p_1 is in P_4, a similar argument follows.
2. p_2, p_3, and p_4 are in $coreSet_2$. Assume that an extension in which p_4 later outputs 0

exists. Then it must be the case that p_3 starts with input value 0. Since p_1 can't hear from both p_3 and p_4 before later outputting 1, it must hear from p_2. If it hears from p_2 the messages of the form $\langle 4_0 \cdot 4 \cdot 2 \rangle$ and $\langle 3_0 \cdot 4 \cdot 2 \rangle$, and $\langle 4_0 \cdot 3 \cdot 2 \rangle$ and $\langle 3_0 \cdot 3 \cdot 2 \rangle$, it cannot later output 1 (since necessarily p_2 and another party in $coreSet_4$ must have forwarded more than one initial value message with value 0). So then p_2 must send to p_1 the messages necessary for p_1 to output 1 before it sends those messages. But if it does that, p_4 would hear all of those messages and eventually have more than one party in its set $coreSet_4$ that forward more than one initial value message containing 1, a contradiction.

□

Theorem 92. *The protocol in Figure 3.2 solves Byzantine fault tolerant BCA for $n = 7$ parties and $f \leqslant 2$ in 3 communication rounds without a PKI setup.*

Proof. As in the previous theorem, all that is left to show is that the protocol is binding.

Binding. We use a proof by contradiction. Consider the first nonfaulty party to output, p_*. Once p_* outputs, there must be an extension in which a nonfaulty party p_1 outputs 1 and an extension in which a nonfaulty party p_0 outputs 0. We refer to the extensions as ext-1 and ext-0, respectively. Assume w.l.o.g. that a majority of the parties in $coreSet_*$ ($\geqslant 3$) sent input value messages containing 1. Then p_0 cannot be in $coreSet_*$. This follows from two points: the fact that p_* outputs before p_0 does and the condition on line 20 by which a party decides a non-$\perp$ value. Let $support_l$ for $l \in \{0, 1\}$ be a set of 3 distinct parties from $coreSet_l$ at the time at which p_l decides such that $\forall p_j \in coreSet_l$, $|S_j| \leqslant 2$, where $S_j = \{s | s_{1-l} \in initVals_j\}$ (note that $coreSet_l$ must contain at least 3 parties satisfying this condition for p_l to decide l). $coreSet_0 \cap coreSet_*$ contains at least 3 parties, at least one of which must be in $support_0$. We consider 3 possible cases:

1. There is a single party in $support_0 \cap coreset*$, and it is honest. Refer to this party as p_H. It must send all of its messages to p_0 that are necessary for p_0 to output 0 before it forwards the initial messages of all parties in $coreSet_1$ (otherwise it cannot be in $support_0$). Therefore, it must receive the messages where all parties in $coreSet_0$

forward the initial messages of all of the parties in $coreSet_0$. At least 3 parties in $coreSet_0$ must also be in $coreSet_*$. Note that then p_1 cannot be in $coreSet_0$, and p_1 must be in $coreSet_*$. p_* expects all parties in $coreSet_*$ to forward all of the 2-chain messages sent by parties in $coreSet_*$, contradicting that p_1 could output 1 after p_* outputs by the condition on line 20.

2. There is a single party in $support_0 \cap coreset*$, and it is Byzantine. Refer to this party as p_B. In ext-0, p_0 must not hear all of the messages where p_B forwarded the input value of all parties in $coreSet_*$, since $f + 1 = 3$ of those parties have input value 1 and in that case, p_B can't be in $support_0$. In order not to contradict Lemma 90 in ext-0, p_B must forward all of the input value messages of parties in $coreSet_0$ prior to forwarding all of the input value messages of parties in $coreSet_*$. In addition, all honest parties in $coreSet_0 \cap coreset*$ (there must be at least 1) must forward all messages that they need to send p_0 in ext-0 prior to forwarding the messages where p_B forwards the input value messages of all parties in $coreSet_*$ (otherwise p_0 hears that p_B forwarded 3 initial value messages with value 1, and it waits to receive the corresponding 2-chain messages from p_B prior to outputting in ext-0, a contradiction). Refer to such an honest party in $coreSet_0 \cap coreset*$ as p_{h0*}. To forward all messages that they need to send p_0 in ext-0 prior to forwarding the messages where p_B forwards the input value messages of all parties in $coreSet_*$, p_{h0*} must receive messages from all parties in $coreSet_0$ forwarding the initial value message of all parties in $coreSet_0$. This implies that p_1 cannot be in $coreSet_0$, and by quorum intersection it must be in $coreSet_*$. The rest of the proof follows the same as that of case 1.
3. Both parties in $support_0 \backslash p_0$ are honest and in $coreSet_*$. Note that the parties must send all messages that they need to send to p_0 in ext-0 prior to forwarding the initial messages of all parties in $coreSet_*$ (as they cannot be in $support_0$ if p_0 hears them forward the initial value messages of three parties with value 1). To do so, they need to forward the initial messages of all parties in $coreSet_0$, as well as the messages in which every party in $coreSet_0$ forwards the initial message of every party in $coreSet_0$. This means that they must receive those 1-chain and 2-chain messages from each

party in $coreSet_0$ (implying that p_1 cannot be in $coreSet_0$). By quorum intersection, it must be the case that there is at least one party in $support_1 \cap coreSet_0$. Let this party be p_{s1c0}. Note that p_1 cannot be in $coreSet_*$ since there are at least 3 parties in $coreSet_0 \cap coreSet_*$, $p*$ hears from the parties in $support_0 \backslash p_0$ that they forwarded all of the initial messages of parties in $coreSet_0$, and p_* expects all parties in $coreSet_*$ to forward all 1-chain and 2-chain messages sent by these parties. Since neither p_1 nor p_0 are in $coreSet_*$, it must be the case that $p_{s1c0} \in coreSet_*$. In order to output, p_*, by the conditions on line 18, requires all parties in $coreSet_*$ to also forward the messages where p_{s1c0} forwards the initial messages of all parties in $coreSet_0$, and it has to hear these corresponding 2-chain messages from p_{s1c0}. Unless all nonfaulty parties in $coreSet_* \cap coreSet_1$ send all of the messages they need to send to p_1 for it to output 1 before forwarding these messages, ext-1 cannot exist. There must be at least one honest party in $coreSet_* \cap coreSet_1$ and it must receive from all parties in $coreSet_1$ the forwarded initial messages of all parties in $coreSet_1$ to do so. Clearly then, p_0 can't be in $coreSet_1$, so one of the parties in $support_0 \backslash p_0$ must be in $coreSet_1$. This party cannot forward all of the input value messages of all parties in $coreSet_1$ prior to sending to p_0 all of the messages it needs to send for it to output 0 (as then p_0 would hear that a party in $support_0$ forwarded $> f$ initial value messages with value 1). Due to FIFO channels p_1 inevitably hears from this party that p_{s1c0} forwarded the initial messages of all parties in $coreSet_0$, a contradiction.

4. There is one honest party and one Byzantine party in $support_0 \backslash p_0$, and both of them are in $coreSet_*$. Refer to the Byzantine party in this set as p_B and the honest party in this set as p_H. Note that by quorum intersection, one of the parties in the set $\{p_0, p_H, p_B\}$ must be in $coreSet_1$. Using similar reasoning to that of case 3, we first show that it cannot be the case that p_0 or p_H is in $coreSet_1$. p_0 cannot hear the 2-chain messages in which p_H forwards the initial value messages of all parties in $coreSet_*$, but p_H must send these messages prior to p_* outputting. So p_H must send to p_0 all of the messages it needs to send to p_0 in ext-0 for p_0 to output 0 prior to forwarding the initial value messages of all parties in $coreSet_*$. For this to happen, p_H must receive

from all parties in $coreSet_0$ the 2-chain messages in which they forward the input value messages of all parties in $coreSet_0$. By quorum intersection, there must be at least one party, p_{s1c0} in $support_1 \cap coreSet_0$. If p_H is in $coreSet_1$, it notifies p_1 that a party in $support_1$ forwarded 3 input value messages with value 0, a contradiction. By the fact that at least 3 parties in $coreSet_0$ must also be in $coreSet_*$, using similar logic to that used in case 3, p_1 can't be in $coreSet_*$ or $coreSet_0$, so p_{s1c0} must be in $coreSet_*$. Due to FIFO channels, p_* expects all parties in $coreSet_*$ to forward the 2-chain messages where p_{s1c0} forwards all of the input value messages of parties in $coreSet_0$. This implies that all honest parties in $coreSet_1 \cap coreSet_*$ must send to p_1 all of the messages they need to send to p_1 in ext-1 prior to forwarding all 2-chain messages of p_{s1c0}, and prior to p_* outputting. This implies that p_0 cannot be in $coreSet_1$.

We now show that binding cannot be broken if p_B is in $coreSet_1$. As argued above, the honest party in $coreSet_* \cap coreSet_1$ requires all parties in $coreSet_1$ to send it the forwarded initial messages of all parties in $coreSet_1$ before it forwards all messages necessary for p_* to output. So it requires a message from p_B forwarding all initial messages of all parties in $coreSet_1$. Since p_B is in $support_0$, p_0 should not hear this message. Since we have shown that p_1 and p_0 cannot be in each others' *coreSet* or in $coreSet_*$, there are at least 3 parties in $coreSet_0 \cap coreSet_* \cap coreSet_1$ and at least one of them must be honest. This honest party must send all the messages it needs to send to p_0 and p_1 prior to sending p_* all the messages it needs to send p_* to output (otherwise it will notify p_0 that p_B forwarded the initial messages of 3 parties with input 1 or it will inform p_1 that a party in $support_1$ forwarded the initial messages of 3 parties with input 0). If it sends all messages for ext-0 first, it will notify p_1 that a party in $support_1$ forwarded 3 initial value messages containing 0 due to FIFO channels. If it sends all of the messages for ext-1 first, it will notify p_0 that a party in $support_0$ forwarded 3 initial value messages containing 1. Either way, we have arrived at a contradiction.

5. There are two Byzantine parties (p_{B1} and p_{B2}) in $support_0 \backslash p_0$, and both of them are in $coreSet_*$. By quorum intersection, there must be some honest party, p_h in $coreSet_0 \cap coreSet_*$ that has to send everything for ext-0 to p_0 before it sends all of

its messages for p_* to output, because otherwise it will reveal to p_0 that a party in $support_0$ forwarded 3 input value messages with value 1, a contradiction. Note that this implies that p_* hears that 3 parties in $coreSet_*$ forwarded 3 input value messages with value 0 prior to outputting, and it must hear all parties in $coreSet_*$ forward these messages. Hence, p_1 cannot be in $coreSet_*$. p_1 also can't be in $coreSet_0$ since p_h expects to hear from all parties in $coreSet_0$ that they forward the input value messages of all parties in $coreSet_0$. Since p_{B1} and p_{B2} are in $coreSet_*$, and p_0 forwards the initial value messages of all parties in $coreSet_0$ and cannot hear 3 input value messages with value 1 before p_* outputs for ext-0 to exist, Lemma 90 implies that $support_1 \cap support_0 = \varnothing$. Thus, the two parties in $support_1 \notin p_1$ must be honest and in $coreSet_*$. By quorum intersection, some party in $support_1$ must be in $coreSet_0$, some party in $support_0$ must be in $coreSet_1$, and there must be an honest party $p_{h,int} \in coreSet_1 \cap coreSet_* \cap coreSet_0$. As already noted, all honest parties in $coreSet_0 \cap coreSet_*$, including $p_{h,int}$, must send all messages they need to send in ext-0 prior to sending all messages for p_* (and thus revealing that a party in $support_0$ forwarded 3 1s). To do so, $p_{h,int}$ will send messages that a party in $support_1 \cap coreSet_0$ (there must be at least 1) forwarded 3 0s; but this should not be revealed to p_1 before it outputs. This means that $p_{h,int}$ should send all messages for ext-1 before sending all messages for ext-0, but then it would reveal to p_0 that a party in $support_0$ forwarded 3 1s. Either way, ext-1 and ext-0 cannot both be possible when p_* outputs, and binding cannot be broken.

□

3.4 Generalizing the Lower Bounds

In this section, we generalize the lower bounds from lower bounds specifically for $n = 3, n = 4$ or $n = 7$ to lower bounds for $n \geqslant 3, n \geqslant 4$ or $n \geqslant 7$. The techniques for generalizing the lower bound in the case that $n \geqslant 3, n \geqslant 4$ are standard and provided for completeness. On the other hand, generalizing the lower bound for $n \geqslant 7$ is slightly more intricate. In the following we simply show how to generalize two of the lower bounds presented above, but

generalizing the other ones (with different corruption models or numbers of rounds) is done in the same manner.

We start by showing how to generalize the lower bound for $n = 4$ and $f = 1$ to any n, f such that $4f \geqslant n \geqslant 3f + 1$. Identical arguments can be made to generalize the lower bounds for $n = 3$ and $f = 1$ to any n, f such that $3f \geqslant n \geqslant 2f + 1$.

Theorem 93. *Assume that it is impossible to solve Byzantine fault tolerant crusader agreement in two rounds with $n = 4$ parties and $f = 1$ faults. Then it is impossible to construct such a protocol for any $f \in \mathbb{N}$ and $4f \geqslant n \geqslant 3f + 1$.*

Proof. Assume by way of contradiction, that for some f, n such that $4f \geqslant n > 3f$ there exists a Byzantine fault tolerant crusader agreement protocol for n parties resilient to f corruptions in which all parties decide on a value after at most two rounds without a PKI setup. We will use this protocol to construct a Byzantine fault tolerant crusader agreement protocol for 4 parties with 1 corruption that requires the same number of rounds, contradicting the theorem statement.

The protocol is designed for 4 parties $p'_1, \ldots, p'_4$ which simulate a full run of the n-party protocol running with parties $p_1, \ldots, p_4$. Start by partitioning the parties $p_1, \ldots, p_n$ into 4 roughly-equal groups: $P_1, \ldots, P_4$. Since n is not necessarily a multiple of 4, it is possible that some of the groups will contain one more party than the other groups. More precisely, set $\ell = (n \mod 4)$, and let $P_1, \ldots, P_\ell$ be of size $\lceil \frac{n}{4} \rceil$ and $P_{\ell+1}, \ldots P_4$ be of size $\lfloor \frac{n}{4} \rfloor$. In case that $\ell = 0$, this means that all set are exactly of size $\frac{n}{4}$. Note that in all other cases, this means that the sets do indeed contain a total of n parties, since their combined sizes are $\ell \cdot \lceil \frac{n}{4} \rceil + (4 - \ell)\lfloor \frac{n}{4} \rfloor = \ell \cdot (\lfloor \frac{n}{4} \rfloor + 1) + (4 - \ell)\lfloor \frac{n}{4} \rfloor = 4 \cdot \lfloor \frac{n}{4} \rfloor + (n \mod 4) = n$.

Now, in the 4-party protocol each party p'_i simulates the full n-party protocol for the parties in P_i. Every party p'_i receives an input x_i and simulates the actions of all parties in P_i after starting with the input x_i. This is done by running the code of each of those parties after receiving that input, and sending messages if required as described below. Whenever p'_i sees that party $p \in P_i$ sends a message m to some party $q \in P_j$ it does the following: if $j = i$, it simulates q receiving m by running the code that q would have run upon receiving

the message from p. Otherwise, p'_i sends the message m to p'_j, along with the information that p sent the message to q. Similarly, when a party p'_j receives a message m from p'_i with the information that $p \in P_i$ sent that message to $q \in P_j$, p'_j simulates q receiving that message by running the code that q would have run upon receiving that message from p. Once p'_i sees that all of the simulated parties in P_i output values, it does the following: if at least one party in P_i output $\perp$, it outputs $\perp$. Otherwise, it outputs some non-$\perp$ value that a party in P_i output[1]. In this setting, the adversary can only corrupt a single party p'_i, which simulates the parties in P_i. The number of parties in P_i is at most $\lceil \frac{n}{4} \rceil$. By assumption, $n \leqslant 4f$, so $\lceil \frac{n}{4} \rceil \leqslant \lceil \frac{4f}{4} \rceil = f$. All other simulated parties act exactly the same as they would when receiving messages in the original protocol, since they are instructed to send and receive messages exactly as they would in the original protocol. In other words, the simulated run perfectly corresponds to a run in which the adversary corrupts at most f parties, in which messages between parties in the same set P_i are delivered immediately and the rest of the messages are delivered according to the scheduling dictated by the adversary. The protocol is secure under these conditions, and thus Validity, Agreement and Termination hold in the simulated run.

In order to complete the proof, all that is left to show is that the resulting 4-party protocol is a two-round Byzantine fault tolerant crusader agreement protocol with $n = 4$ and $f = 1$, reaching a contradiction to the theorem statement.

Validity. If all parties have the same input b, then each nonfaulty p'_i simulates all of the parties in P_i with the input b. This means that the run corresponds to a run in which all parties simulated by nonfaulty parties have the input b. From the Validity property of the original protocol, all simulated nonfaulty parties output b as well, and thus every nonfaulty p'_i output b after seeing that all of the parties in P_i output that value.

Agreement. Assume that two nonfaulty parties p'_i and p'_j output the non-$\perp$ value b_i and b_j respectively. Before doing so, each one saw that all of the parties simulated by it completed the protocol and that at least one of the parties simulated by p'_i and p'_j output b_i

[1] An alternative choice is to output $\perp$ only if all simulated parties did, and otherwise output some non-$\perp$ value.

and b_j respectively. Those parties are simulated as nonfaulty parties, so $b_i = b_j$ from the Agreement property of the original protocol.

Termination. If each nonfaulty p'_i starts the protocol, it simulates all of the parties in P_i correctly throughout the whole protocol. This means that all of the parties in the P_i sets simulated by nonfaulty parties act as nonfaulty parties would in the original protocol, and thus eventually decide. After seeing that all of the parties in P_i output some value, every nonfaulty p'_i outputs a value as well.

Round Complexity. In the original n-party protocol, all parties output a value after two rounds. More precisely, all nonfaulty parties send only round 1 or round 2 messages. Observe a given run of the 4-party protocol. In the simulated n-party protocol, all simulated parties output a value after at most 2 rounds without sending any message from round 3 or higher. Therefore, in the 4-party protocol, no party sends a message from round 3 message or higher, and after every nonfaulty simulated party decides a value, every nonfautly p'_i outputs a value as well. □

Theorem 94. *Assume there is a network of 7 parties $p_1, \ldots, p_7$, and let S, T be a partitioning of the parties such that $|S| = 4, |T| = 3$, $S \cup T = \{p_1, \ldots, p_7\}$ and $S \cap T = \varnothing$. Assume that it is impossible to solve Byzantine fault tolerant binding crusader agreement in two rounds with $n = 7$ parties and $f = 2$ faults, even if the adversary can corrupt at most one party in S and one party in T. Then it is impossible to construct such a protocol for any $f \geqslant 2$ and $4f > n > 3f$.*

Proof. Assume by way of contradiction that such a protocol exists for some n, f such that $f \geqslant 2$ and $4f > n > 3f$. The proof follows a similar outline to the previous proof, simulating the n party protocol in the 7 party setting. Without loss of generality, assume that $S = \{p_1, \ldots, p_4\}$ and that $T = \{p_5, \ldots, p_7\}$. Since $4f > n > 3f$, there exists some $k \in [f - 1]$ such that $n = 3f + k$.

We will now construct a protocol for 7 parties $p'_1, \ldots, p'_7$. Start by partitioning the parties $\{p_1, \ldots, p_n\}$ into 7 sets $P_1, \ldots, P_7$. Each set in $P_1, \ldots, P_4$ contains k parties for the k defined above, and each party in $P_5, \ldots, P_7$ contains $f - k$ parties such that for every $i \neq j$, $P_i \cap P_j = \varnothing$. First, note that by definition $f > k > 0$ and thus also $f > f - k > 0$. This

means that each of these sets has a positive number of parties, smaller than f. In addition, the total number of parties is $4 \cdot k + 3 \cdot (f - k) = 3f - 3k + 4k = 3f + k = n$. In other words, it is possible to partition the n parties into non-intersecting sets of these exact sizes.

From this point on, the simulation is exactly the same as in Theorem 93. Each party p'_i is in charge of simulating the parties in P_i. It starts the protocol by receiving its input x_i and simulating all of the parties in P_i starting the protocol with the same input x_i. Following that, if some simulated party $p \in P_i$ sends a message m to $q \in P_j$ it either delivers it immediately if $i = j$ or sends m to p'_j and signifies that p sent the message to q. Upon p'_j receiving a message m from p'_i saying that p sent that message to q, p'_j checks that $p \in P_i$ and $q \in P_j$. If that is the case, p'_j simulates q receiving that message from p. In all of the above discussion, by "simulating receiving the message" we mean that the simulating party runs the code that the simulated party would have run, and sends any messages according to the above description.

Once p'_i sees that all of the parties in P_i output some value, it outputs if at least one of the parties in P_i output $\perp$, p'_i outputs $\perp$ as well. Otherwise, it outputs some non-$\perp$ value that a party in P_i output. All that is left to do, is to show that the protocol is a 2-round protocol, resilient against a Byzantine adversary that controls at most one party in S and one party in T, reaching a contradiction. An adversary controlling at most one party in S and one party in T is in charge of simulating at most $f - k + k = f$ parties. This means that any run of the 7-party protocol corresponds to a run of the n-party protocol in which the adversary controls at most f parties, and the scheduling is the same as the one described in Theorem 93. Therefore, the simulated run terminates in two rounds and has the Validity, Agreement, Termination and Binding properties.

The proof that the 7-party protocol requires two rounds and that it has the Validity, Agreement and Termination properties is identical to the proof in Theorem 93 and is thus omitted. For the final property, Binding, assume some nonfaulty party p'_i outputs some value. At that point in time, it saw that all of the parties in P_i output values. All of those parties are nonfaulty, and thus from the Binding property of the n-party protocol, at that time there exists some value $b \in \{0, 1\}$ such that all nonfaulty parties output either b or $\perp$

in the n-party protocol. We will show that all nonfaulty parties output either b or $\perp$ in the 7-party protocol. Observe some nonfaulty party p'_j in the 7-party protocol. If it outputs the value $\perp$ from the protocol, the property holds. Otherwise, it output some value b' after seeing that at least one party $p \in P_j$ output b', and no party in P_j output $\perp$. From the Binding property of the n-party protocol, $b' = b$, and thus p'_j outputs b as well. $\square$

4. Dynamically Available Consensus Under Generalized Synchrony

This work was done in collaboration with Kartik Nayak (Duke University), Nibesh Shrestha (Supra Labs), and Luca Zanolini (Ethereum Foundation), with Sravya Yandamuri as lead author.

Byzantine fault-tolerant (BFT) Total-Order Broadcast (TOB) enables a group of parties to agree on a sequence of client transactions despite the presence of a bounded number of Byzantine (malicious) parties. The traditional model assumes a permissioned network with a fixed set of parties known to each other. Bitcoin [71] introduced the first protocol operating in the *dynamically available* model [59], where parties can go online or offline freely, and the total number of active participants is unknown. Pass and Shi later formalized this concept as the sleepy model of consensus [72]. Dynamic availability is a crucial property for blockchain protocols, ensuring both safety and liveness even when an external adversary forces a large fraction of parties offline or introduces a surge of new parties.

There is a long line of work in improving the latency of dynamically available protocols [50, 7, 45, 86]. Initial works based on longest-chain-based protocols incurred latency that depended on the security parameter and the actual level of participation. Several works, including Goyal et al. [50], Prism [7], parallel chains [45], and OHIE [86], removed the dependency on one of these parameters but not the other. Finally, MR [68] and MMR [65, 64] designed protocols with $O(\Delta)$ latency where Δ is a pessimistic network delay. However, all of these protocols only work under a synchronous [35] network assumption. While it is conceivable that the network is synchronous under a very large value of Δ (e.g., 10 mins), it may be the case that the maximum bound on the actual network delay is much lower. At the same time, relying on synchrony implies that the latency is a function of this pessimistic parameter Δ. On the other hand, while partially synchronous [38] and asynchronous protocols can be designed to allow parties to commit at the speed of the actual network delay, dynamically available protocols are known to not be secure under partial synchrony or asynchrony [59]. Thus, we have the following natural question:

Can we obtain the best of both worlds, i.e., be secure under the synchrony assumption

yet commit in time that is closer to the actual network delay?

We answer this question affirmatively in this work through two key contributions. First, we introduce a new network model called *Generalized Synchrony*, which serves as an intermediate assumption between synchrony and partial synchrony. Second, we design a dynamically available protocol that remains secure under the generalized synchrony network assumption.

Generalized synchrony. The generalized synchrony model can be thought of as an intermediate between synchrony and partial synchrony. In this model, a protocol designer chooses a sequence of k values $\Delta_1, \ldots, \Delta_k$ such that $\forall i, \Delta_i < \Delta_{i+1}$ and the actual network delay is always less than some Δ_j for $j \in [1, k]$ (we say Δ_j *holds*). We refer to the smallest such Δ_j value that holds as Δ_{ideal}. The synchronous network can be thought of as a specific case of generalized synchrony where $k = 1$ and Δ_1 always holds. Generalized synchrony also bears a resemblance to the *unknown* Δ approach in partial synchrony — typically protocols secure in this version attempt to achieve consensus under increasing values of Δ without knowing which value will hold. In comparison, under generalized synchrony, it is known that Δ_k always holds.

Dynamic availability under generalized synchrony. Our core contribution is the design of a dynamically available Total-Order Broadcast protocol under generalized synchrony, which eventually commits with a latency of $O(\Delta_{ideal})$ in expectation. We work within the extended sleepy model of Malkhi et al. [65], referred to as the (T_f, T_b, α)-sleepy model, where the total number of corrupt parties during interval $[t - T_f, t + T_b]$ is less than α times the number of awake parties at time t. When $T_f = \infty$, the protocol does not allow the number of corrupt parties to decrease. In particular, we obtain the following result:

Theorem 95. *There is a Total-Order Broadcast protocol under generalized synchrony with parameters $\Delta_1, \ldots, \Delta_k$ in the $(\infty, 20\Delta_k, \frac{1}{2} - \epsilon)$-sleepy model in which blocks are eventually committed with a latency of $O(\Delta_{ideal})$ time in expectation, where $\Delta_{ideal \in [k]}$ is the lowest delay that holds.*

In comparison, the state-of-the-art dynamically available protocols incur a latency of $O(\Delta_k)$ in expectation. In our protocol, if we choose the apriori Δ_i values appropriately (e.g., in powers of two), this latency will be close to the one based on the actual network delay

(e.g., within a factor of 2).

4.0.1 Approach and Key Challenges

Our core idea is to run multiple instances of the expected constant round dynamically available protocol (such as [64]), each based on different assumptions for the upper bound of network delay, denoted as $\Delta_1, \ldots, \Delta_i, \Delta_{i+1}, \ldots, \Delta_k$, where $\forall i \Delta_i < \Delta_{i+1}$. Allowing all instances to propose new blocks simultaneously would lead to conflicting blockchains across different instances. Thus, only one instance is ever allowed to propose new blocks at a time. Initially, this responsibility lies with the instance that has the smallest delay bound, Δ_1. When a block B proposed in a lower-delay instance (e.g., Δ_i) is confirmed in that instance, π_i, it is then passed to the next higher instance, π_{i+1}, for reconfirmation. However, if the lower delay bound (such as Δ_1) does not hold, the protocol may stop receiving new input blocks, or parties may produce inconsistent outputs in that instance. In such cases, the responsibility for proposing new blocks shifts to the next higher instance, and participants stop engaging in the failed instance and all lower instances.

When taking the above approach, we want to ensure that blocks are committed with a latency of $O(\Delta_{ideal})$. To achieve this, we need to address the following key challenges. First, parties should never propose blocks in instances past π_{ideal}. Second, if the parties are proposing in instances π_j for $j < ideal$, they still make sufficient progress. Finally, we need to ensure that new parties joining the protocol have sufficient state to not stall the protocol.

Challenge 1: Ensuring blocks are not proposed in instances past π_{ideal}. Since not all Δ_j hold, it is necessary that the parties notice when sufficient progress is not being made in the sub-protocol corresponding to their estimate of the lowest delay that holds. Thus, if a party observes lack of progress or forks in the chain, then parties must attempt to move to a higher sub-protocol when observing this evidence. We need to, however, ensure that parties do not *incorrectly* attempt to move to a higher sub-protocol. In particular, in our protocol, the parties can only propose and vote for new blocks in a single sub-protocol at a time (otherwise, this would lead to conflicting chains). So even if a party is proposing and voting for new blocks in π_{ideal}, it may not see progress if all of the other awake honest

parties are not also proposing and voting for new blocks in π_{ideal}, as the leader's proposal is not guaranteed to succeed if all of the awake honest parties do not vote for it. Thus, it is necessary then that upon updating their estimate of Δ_{ideal}, parties only start to check that progress is made once they are sure that all of the other awake honest parties have also updated their estimate to Δ_{ideal}. Toward this end, our protocol uses a beacon protocol, which the parties use to locally decide when to increase their estimate of Δ_{ideal}. The beacon protocol is run with a delay of Δ_k, which is guaranteed to hold. Furthermore, it has the property that when one party updates their estimate of Δ_{ideal} to some Δ_j due to outputting j from the beacon protocol, all others will also output j and update their estimates accordingly within $3\Delta_k$. With this knowledge, the parties may safely begin expecting, and checking for, progress in π_j after a $3\Delta_k$ grace period elapses once they update their estimate and ensure that they do not move past π_{ideal}.

Challenge 2: Ensuring progress at the rate of Δ_{ideal}. A second challenge in designing such a protocol is that it is possible that a subset of the honest parties may see evidence that a given Δ_j does not hold, while others may not. For example, it is possible for some honest parties to see progress in $\pi_{j<ideal}$, while other honest parties see no progress. Or, an honest party might see two conflicting blocks decided in π_j, but given that the protocol has fluctuating participation, it may not be able to produce a proof to convince other honest parties that two conflicting blocks were certified. This can potentially ensure that parties are stuck on $\pi_{j<ideal}$ and not making sufficient progress. To handle this challenge, we ensure that if the honest parties *do not* eventually converge on Δ_{ideal}, then this is because progress is being made *at least* at the speed of Δ_{ideal}, and consistent chains are being created in all $\pi_{j\geqslant ideal}$. To achieve this, we have parties check that a block decided in π_j must be decided in π_{j+1} sufficiently quickly; otherwise this is evidence that Δ_j does not hold, and they attempt to increase their estimate. If $j < ideal$, and an honest party does *not* meet this condition, if we set each delay to be double the previous delay, then the parties will make progress *at least* at a rate of 1 block per $O(\Delta_{ideal})$.

Challenge 3: Ensuring newly joining parties do not stall the protocol. A third challenge in designing our protocol relates to parties joining the network. As described in the preceding

paragraphs, there are various criteria that the parties use to detect whether their current estimate of Δ_{ideal} is wrong. The liveness of the protocol requires that if progress is not made in π_j, the parties update their estimates and try to start proposing and voting for new blocks in π_{j+1}. In a network without fluctuating participation, once all of the honest parties notice that Δ_j does not hold, they can all indicate this in the beacon protocol and subsequently update their estimates. However, when parties may join the network at any moment, we must be careful that parties joining at critical moments do not prevent the parties from increasing their estimates when needed, resulting in the protocol losing liveness. This could happen because a party must observe a lack of progress over a sufficiently long time period, and only at the end of that time period can it be sure that its current estimate does not hold. In our work, this is a 2κ epoch time period, where κ is a security parameter. If a newly joined party waits for a 2κ epoch time period to elapse without seeing progress before attempting to increase its estimate of Δ_{ideal}, once it nears the end of this period, *another* honest party may join the protocol, starting another waiting cycle. In this way, the network could never reach a state in which all of the awake honest parties are convinced that progress is not being made, and this could prevent them from ever succeeding in increasing their estimates. To overcome this challenge, we present a joining protocol in which parties verify the amount of progress that has been made in the protocol up to the time they joined. A joining party determines when the first parties to adopt this estimate did so, and verifies if sufficiently many blocks were committed since that point. We ensure that if a party joins the network with the estimate that $j = ideal$ and it is *not* convinced that its estimate is wrong, this is because an average of one block per $O(\Delta_j)$ time steps have been committed since the time at which the first honest parties to set their estimates to Δ_j did so. Otherwise, the party joins the network with its input to the beacon protocol already indicating that it wants to increase its estimate. We overcome these challenges and present a joining protocol that ensures that new parties joining the protocol do not prevent the liveness.

4.1 Technical Overview

As described in the introduction, our work takes the approach of running one instance of the expected constant round dynamically available protocol such as MMR [65], one for each value of Δ_i. Initially, parties estimate that the network delay is Δ_1, considering it to be the actual delay Δ_{ideal}. They adjust this estimate to Δ_2 only when sufficient evidence suggests otherwise. Such evidence typically indicates possible network asynchrony or partitioning, which could lead to inconsistent outputs, in line with the known impossibility result presented in [57]. Eventually, without knowing Δ_{ideal}, we need to ensure that the current estimate $\Delta_j < \Delta_{ideal}$ makes sufficient progress, or the parties will change their estimate to Δ_{ideal}.

Protocol with a single Δ that holds. To begin with, we outline the scenario where $k = 1$, i.e., there is a single delay Δ_1 that holds. The protocol progresses through a series of *epochs*, with each epoch having a designated leader responsible for proposing a new block. The protocol uses a Lock-Commit paradigm; in this case, once a party decides a block (in a chain)[1], all honest parties *lock* onto the same block. The lock ensures safety across epochs: parties are restricted to voting only for blocks consistent with their current locks. This ensures that once a party decides on a block, no other party can subsequently decide on a conflicting block. To maintain the protocol's liveness, it is crucial that honest leaders propose blocks that all honest parties will *vote* for — specifically, blocks that extend the current locks of all honest parties. This ensures that the protocol continues to make progress while preserving safety.

To instantiate the above outline in the dynamically available setting, we adopt the approach of MMR [65]. Each epoch starts with two consecutive instances of Graded Agreement (GA) (Definition 96), where parties choose their inputs to the second GA from their outputs to the first GA. Ultimately, this block of two GA's is used for parties to update their locks (to ensure that honest parties do not have conflicting locks) and choose their candidates which they will extend or propose to ensure that parties propose blocks that all

[1] Since each block uniquely determines its corresponding log, deciding a block B is equivalent to deciding on the log it represents.

honest parties will vote for. To the first GA instance of the epoch, which we denote as GA_0, each party inputs their current lock. By choosing their input in this way, a party ensures that safety remains intact in this protocol, i.e., when one honest party decides a block, no honest party decides a conflicting block. Each party then selects the highest grade 1 output from GA_0 that does not conflict with any other outputs as their input to GA_1, the second Graded Agreement instance. The protocol ensures that such an output is guaranteed to exist by the validity and integrity of GA. Furthermore, the graded delivery property of GA ensures that the inputs to (and outputs from) GA_1 of the honest parties do not conflict. The output from GA_1 is used by honest parties to update their locks. As mentioned before, when an honest party decides on a block, all honest parties lock on it. The validity property of GA ensures that all honest parties output with grade 1, the highest common ancestor of their inputs, and the integrity property ensures that any block output by an honest party must have been input by an honest party (or a block extending it must have been). In this way, the parties choosing to input their locks to GA_0 preserves safety because if all honest parties were locked on a given block at the beginning of the epoch, their locks after the update would continue to extend that block. This mechanism also ensures that the locks of all honest parties are consistent, a crucial requirement for liveness, as parties will only vote for blocks that do not conflict with their current locks.

After outputting from GA_1, in addition to updating their lock, a party selects their *candidate* block as the highest output with any grade. Due to the graded delivery property of GA and the fact that the outputs from GA_1 do not conflict, this process ensures that the candidate blocks chosen by all honest parties are extensions of the locks held by all honest parties. Subsequently, each party generates a new block proposal extending their candidate and submits it to the GPE mechanism (Definition 97). With a probability of $1/2 - \epsilon$, all honest parties will decide on the block proposed by one of the honest parties. If any party decides a block B, all honest parties lock on that block, ensuring safety and liveness.

Extending to the generalized synchrony model. What has been described so far outlines a protocol that functions effectively when a single delay bound, Δ_1, holds. However, in the generalized synchrony model, there are multiple potential delays, $\Delta_1, \ldots, \Delta_k$, and the

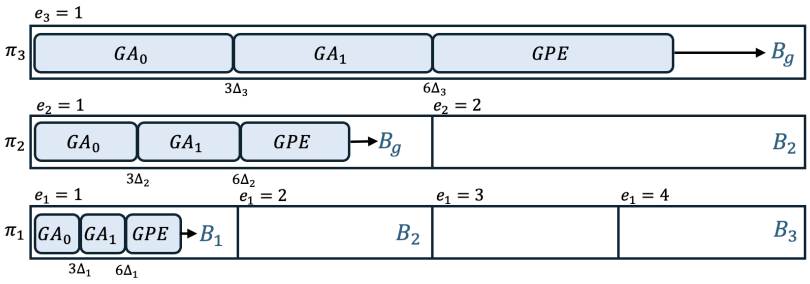

FIGURE 4.1: A depiction of an execution of the protocol from time $t = 0$ to $10\Delta_3$. Each sub-protocol is executed with an increasing Δ, where $\Delta_i = 2\Delta_{i-1}$. Because Δ_1 holds, new blocks are only decided in π_1 and blocks decided in π_1 may be used as input to π_2, and subsequently decided there. Similarly, blocks decided in π_i may be used as input, and decided in, π_{i+1}. In epoch 3 of π_1, no block is decided due to a dishonest leader of GPE.

challenge is ensuring that the protocol makes progress at the speed of the smallest Δ that holds (Δ_{ideal}), without the parties knowing in advance which Δ is the actual one. The goal is to dynamically adjust to the correct delay, enabling the system to progress efficiently while maintaining safety, despite the uncertainty about which Δ governs the current network conditions. Initially, parties assume that the network delay is Δ_1, considering it to be the actual delay Δ_{ideal}. They adjust this estimate to Δ_2 only when sufficient evidence suggests otherwise. Such evidence typically indicates possible network asynchrony or partitioning, which could lead to inconsistent outputs, in line with the known impossibility result presented in [57]. Eventually, either one of two possibilities occurs: (1) the parties converge on the actual network delay, Δ_{ideal}, which corresponds to Δ_k in the worst-case scenario; or (2) if the parties do *not* converge on Δ_{ideal}, this is because progress is being made *at least* at the desired rate of one block per $O(\Delta_{ideal})$ time. Regardless of which case occurs, parties never definitively know which delay is Δ_{ideal}, but the protocol ensures that progress is made at this speed eventually. Importantly, our protocol design ensures that no party increments their estimate past *ideal*, ensuring that progress is made at the speed of Δ_{ideal} and that the prefix property is not violated.

Our high level approach is as follows. The protocol consists of k sub-protocols π_i for $1 \leqslant i \leqslant k$, each associated with a delay parameter Δ_i. We set the delays such that $\Delta_i = \frac{\Delta_{i+1}}{2}$. Participants execute all k sub-protocols simultaneously, continuously updating their estimate of the lowest delay that holds. Once a party has seen evidence that Δ_j does not hold, a

party ceases participation in protocols $\pi_{1,\ldots,j}$. Of course, having k sub-protocols implies that honest parties output k logs. So how would parties choose which log is the *right one*? Our solution is to have each party maintain an *estimate* of the delay that they *believe* is the lowest delay that holds. Parties only accept and propose *new* blocks in the single sub-protocol that is being executed at the delay which they believe is the lowest that holds. In the sub-protocols with higher delays, they only decide on blocks which were already decided in lower sub-protocols.

To determine whether to increment their estimate of Δ, parties rely on three key criteria. The first criteria (criterion 1) checks for the *lack of progress in the current sub-protocol*. Within the sub-protocol, since honest leaders are chosen by the GPE instance with a probability of $1/2 - \epsilon$, we cannot guarantee progress in a given epoch. However, because the leader selection events across instances are independent, we can expect to commit κ blocks within 2κ epochs with high probability. This check is triggered only after all parties have moved to the current sub-protocol (ensuring which requires additional care as discussed in challenge 1 in the previous section). The second criteria (criterion 2) checks for the *lack of confirmation from sub-protocols with higher* Δ. This check ensures that higher sub-protocols, and in particular π_{ideal}, are reconfirming sufficient blocks from the current sub-protocol. Since any $\Delta_{j'} > \Delta_j$ is a stronger assumption, lack of reconfirmations in any higher sub-protocol $\pi_{j'}$ implies insufficient progress in π_j. Finally, the last criteria (criterion 3) checks for *consistency violations using higher sub-protocols*. Again, by a similar reasoning as the previous one, a sub-protocol with a larger Δ is more authoritative. Thus, if, at any point, there is a conflict between what was confirmed in the current sub-protocol and what is reconfirmed in a sub-protocol with a larger Δ, the party will decide to move to the next sub-protocol.

Finally, even if all honest parties have observed evidence to move to a higher sub-protocol, how do they agree to do so? Observe that this evidence suggests that the current estimate of Δ is incorrect; thus, executing any protocol in this instance may not provide with any guarantees. Thus, the parties rely on a beacon protocol π_{beacon} that continuously runs GAs sequentially with parameter Δ_k. The beacon protocol ensures that if some party decides to update its estimate at a given time, all honest parties will update their estimate within

another $3\Delta_k$ time.

Allowing parties to join the protocol correctly. A dynamically available protocol allows parties to join the protocol freely. From a modeling standpoint, we can classify parties in the protocol as being in one of three states: awake (i.e., actively participating parties), asleep (i.e., not participating), and recovering (i.e., a state where they receive sufficient state to be awake). Thus, when a party joins, it first enters the recovering state, then engages in a *joining protocol* to learn the necessary state required to participate before transitioning to the awake state. A natural question is, how does the party learn this necessary state in the joining protocol?

In the generalized synchrony model, a naïve approach is for joining parties to start with their estimate $est = 1$ (like the initial awake parties) and the necessary information to participate in each sub-protocol (i.e., a log and a lock). However, the protocol's liveness depends on parties incrementing est when progress stalls in the sub-protocol proposing new blocks. In particular, we need all awake honest parties to propose and vote for blocks in the same sub-protocol. Thus, if new parties continuously join with $est = 1$, the protocol may *never* advance beyond π_1, even when Δ_1 does not hold. This is because parties rely on the beacon protocol to move to higher sub-protocols, and the GA invocation in the beacon protocol is guaranteed to suggest moving only if all honest parties input an intent to move. Since a joining party does not have access to this information, if it indicates a lack of this intent, the parties may not be able to move to higher sub-protocols. Thus, if parties keep joining the network, the parties may never move past $est = 1$. On the other hand, if the joining party suggests an intent to move incorrectly (i.e., when the estimate is correct), this may eventually cause parties to move beyond π_{ideal}.

A better approach is for joining parties to first obtain the outcome of an iteration of the beacon protocol before joining. This allows them to determine the appropriate est value to start with. However, the same issue persists. The already awake parties may have noticed insufficient progress in the current sub-protocol which a joining party would not know. Just as it is about to set its intent to move, another party might join, stalling progress again. Thus, to prevent perpetual stalling at critical moments, parties must only join with an

intent to not move if sufficient progress has been made in the current sub-protocol up to their joining time.

This leads to our solution, which relies on the following key idea: A party joining with a given est with a lack of intent to move should do so only after verifying that sufficient progress has been made since the time that parties first adopted this value of est. To achieve this, the joining party must not only learn est from π_{beacon} but also obtain t^*_{est}, which represents the time when the first honest parties that adopted this value of est did so. Additionally, the joining party must acquire a log for each sub-protocol that matches the log of some already awake honest party, preventing the adversary from falsely inflating the perceived progress. Using t^*_{est} and these logs, the joining party can verify whether enough blocks have been committed in π_{est} without triggering criterion 1. Moreover, it can determine if those blocks were committed in higher sub-protocols as expected to avoid triggering criterion 2, and whether the logs conflict with the ones in higher sub-protocols triggering criterion 3. Using this approach, our protocol ensures that the joining party assumes the approximate state of some awake honest party after recovering.

4.2 Model and Definitions

We consider a system with n *parties*, denoted as $\mathcal{P} = \{p_1, \ldots, p_n\}$, connected through pairwise reliable, authenticated point-to-point channels. The system operates under an adaptive adversary capable of corrupting parties at any point during execution. Byzantine parties may exhibit arbitrary behavior, while honest parties strictly follow the protocol as specified.

Cryptographic assumptions. We assume the use of a Public Key Infrastructure (PKI) and digital signatures. A message m sent by party p_i is digitally signed with p_i's private key, denoted as $\langle m \rangle_i$. Additionally, we assume a collision-free hash function, with $H(x)$ representing the invocation of the hash function H on input x. Lastly, we assume a verifiable random function (VRF). Each party p_i with its secret key can evaluate $(\rho, \pi) \leftarrow \mathsf{VRF_i}(^-)$ on any input μ. The output is a deterministic pseudorandom value ρ along with a proof π. Using π and party p_i's public key, anyone can verify whether ρ is the correct evaluation of

VRF_i on input μ.

Time. Our protocols operate in discrete rounds, where each round is represented by a value in $\mathbb{N}$. Protocol execution begins at round 1. We assume that all parties have perfectly synchronized clocks.

Sleepiness. Parties can be in one of three states during the protocol: *awake*, *asleep*, or *recovering*. Parties that are awake actively participate in the protocol, while parties that are asleep neither participate nor receive any messages. A party enters the recovering state when they are engaged in a *joining protocol*[2]. During the recovering state, the party receives all messages sent from the time they begin the joining protocol at time t, and they continue to receive messages until they transition to the asleep state. Importantly, we do not assume that messages are queued for a party while they are asleep.

Our protocol operates in the extended sleepy model of [65], which differs from the original sleepy model of [72]. In this extended model, the participation of corrupt parties may grow proportionally to the overall participation of the network, with a certain delay. Let T_b and T_f be non-negative integers representing two specific time constants and let $\alpha \geqslant 1$ denote a predetermined failure ratio of honest to Byzantine parties. Our protocol works in the (T_f, T_b, α)-sleepy model [65]. Let n_t denote the total number of parties awake at time t, and let $\mathcal{F}_t$ be the total number of corrupt parties awake at time t. Then:

$$f(t, T_f, T_b) = |\bigcup_{t-T_f \leqslant \tau \leqslant t+T_b} \mathcal{F}_t|. \tag{4.1}$$

An execution of the protocol is *admissible* in the (T_f, T_b, α)-sleepy model if for all $t \geqslant 0$ it holds that $f(t, T_f, T_b) < \alpha n_t$. To put it simply, the total number of Byzantine parties awake during the entire time interval from $t - T_f$ to $t + T_b$ must be less than α fraction of the total number of parties awake at time t.

[2] A joining protocol enables parties that were previously offline to retrieve messages sent during their absence. The duration of this protocol typically depends on the volume of data to be recovered. For this discussion, we consider the joining protocol in abstract terms, without delving into its specific mechanics.

4.2.1 Graded Agreement, Graded Proposal Election, and Total-Order Broadcast

Logs and blocks. A *log* is defined as a finite sequence of *blocks*, represented by $\Lambda = [B_1, B_2, \ldots, B_k]$, where each block B_i contains a set of *transactions* and references the preceding block. For this discussion, we assume the existence of an external transaction pool from which honest parties retrieve transactions.[3] Each block includes a hash pointing to the previous block, creating a hash chain. The initial block in this chain, which lacks a reference to any other block, is called the *genesis block* and is denoted as B_g. The *height* of a block is defined by its position in the chain, measured as the distance from the genesis block. Any block B_k specifies a unique chain $B_0, \ldots, B_k$, which corresponds to a distinct log.

Two logs are considered *conflicting* if neither is a prefix of the other. Likewise, two blocks B and B' are said to conflict if the logs they uniquely define are conflicting. A log Λ is described as an *extension* of Λ' if and only if Λ' is a prefix of Λ. If $\Lambda' = \Lambda$, then Λ' and Λ are prefixes of each other. We use the notation $\Lambda[:-c]$ to indicate the last c blocks in the log Λ. We assume the existence of a function max() which takes as input two logs and returns the longer of the two. If both logs have the same length, it returns an arbitrary one. In addition we assume the existence of a function diff() which takes as input two logs and returns the blocks that are only in the longer log and not in shorter log. We make no assumptions on what it returns if neither log is a prefix of the other.

Graded Agreement. In the *Graded Agreement* (GA) primitive, each participant holds a block B, and they output blocks with an associated *grade*, either 0 or 1. We represent the output as a pair (B, g). The log that party p_i submits to the GA is referred to as the *input of party* p_i.

Definition 96. *We require the Graded Agreement primitive to satisfy the following properties.*[4]

Graded delivery. *If an honest party outputs* $(B, 1)$*, then all honest parties output* $(B, *)$*.*

Integrity. *If an honest party outputs* $(B, *)$*, then at least one honest party has input* B' *extending*

[3] These transactions are validated against a set of predetermined rules before being compiled into blocks.

[4] It is important to note that there are several versions of Graded Agreement, each with distinct properties. The protocol presented in this work is based on the versions with the properties specified in [65].

B.

Validity. *Let B be the highest block that every honest party's input extends. Then all honest parties output* $(B, 1)$.

Note that there is no consistency property, so a single party may output multiple conflicting blocks.

Definition 97 (Graded Proposal Election)**.** *In graded proposal election (GPE), each party* propose*s their own block B and the parties* elect *a single block with grades. Each party is equipped with a function* predicate() *that takes as input a block B and outputs 0 or 1. At the end of the protocol, each party* outputs *a single pair* (B, g) *of a block B (or* $B = \perp$*) and a grade* $g \in \{0, 1\}$ *with the following guarantees:*

Consistency. *If two honest parties output* $(B, *)$ *and* $(B', *)$ *for* $B, B' \neq \perp$*, then* $B = B'$.

Graded delivery. *If an honest party outputs* $(B, 1)$*, then all honest parties output* $(B, *)$.

Validity. *If all honest parties input blocks that satisfy* predicate() *for all honest parties, then with probability* $\frac{1}{2} + \epsilon$*, all honest parties output* $(B, 1)$ *where B is the input of an honest party.*

Integrity. *If an honest party outputs* $(B, *)$*, then B satisfies* predicate() *for at least one honest party.*

Throughout this paper, we treat Graded Agreement and Graded Proposal Election as black-box protocols. The GA protocol we utilize requires 3Δ time, while the GPE protocol takes 4Δ time [65]. Our focus is on the specific properties these protocols satisfy, and we build our solution upon those properties.

Total-Order Broadcast [27]. A Total-Order Broadcast (TOB) protocol ensures that all the honest parties *decides on* the same log Λ. A protocol for BFT TOB satisfies the following properties.

Safety. If two honest parties deliver logs Λ_1 and Λ_2, then Λ_1 and Λ_2 do not conflict.

Liveness. For any *valid* transaction tx present in the transaction pool, there is a point in time t such that all honest parties, provided they remain active for an adequate period of time[5], will eventually produce a log Λ containing (within one of its blocks) the transaction tx.

We define a TOB protocol as *dynamically available* if it operates as a TOB protocol within the (T_f, T_b, α)-sleepy model.

4.3 Dynamically Available TOB Under Generalized Synchrony

In this section, we first discuss the generalized synchrony model. We then provide a high-level overview of a protocol for dynamically available Total-Order Broadcast within the generalized synchrony model. In the following subsection, we present the pseudo-code for the protocol and provide a detailed explanation of its execution, highlighting how the protocol operates step by step under various conditions.

4.3.1 Generalized Synchrony

Synchronous protocols require that *all* messages sent by honest parties are delivered within some fixed known time Δ after they are sent by the sender. This model has the advantage of being able to tolerate up to one-half Byzantine faults. In practice, ensuring the delivery requirement implies that the parameter Δ needs to be chosen pessimistically. On the other hand, partially synchronous and asynchronous protocols can tolerate asynchrony in the network, but protocols under these models tolerate only one-third Byzantine faults.

Recent research in distributed consensus has focused on developing Total-Order Broadcast protocols within various adaptations of the sleepy model [72]. This has led to the creation of a series of dynamically available total-order broadcast protocols [65, 25, 26, 68, 46, 27], which are capable of tolerating fluctuations in the participation levels of parties. These protocols must be synchronous [57][59, Theorem 7.1], meaning they rely on a known, system-wide upper bound Δ for communication delay and clock skew, while assuming that local computation time is negligible. This is, in practice, a significant limitation because

[5] The exact length of "adequate time" is protocol-dependent.

even short periods of asynchrony or network partitioning can jeopardize everything done before asynchrony started.

In this work, we introduce a new model called the *generalized synchrony* model. In this model, parties have synchronized clocks, and we assume a *range* of possible network delay values, denoted as $\Delta_1, \Delta_2, \ldots, \Delta_k$, where Δ_1 represents the smallest delay and Δ_k the largest. The *actual* network delay may fluctuate between Δ_1 and Δ_k, with the assumption that it is bounded by Δ_k. The smallest such delay that holds is denoted as Δ_{ideal}, where $\Delta_1 \leqslant \Delta_{ideal} \leqslant \Delta_k$. Message delays are therefore bounded by Δ_{ideal} rounds.

The generalized synchrony model allows us to circumvent some impossibility results pertaining to partial synchrony (e.g., the lower bound in [37]) (since Δ_k always holds). At the same time, it allows us to design protocols that obtain a latency better than the synchronous model (since Δ_{ideal} may be less than Δ_k).

In this new model, we have the following definition.

Definition 98 (Byzantine Fault-Tolerant Total-Order Broadcast Under Generalized Synchrony)**.** *A Byzantine fault-tolerant Total-Order Broadcast protocol within the generalized synchrony model with progressively increasing delays $\Delta_1, \ldots, \Delta_{ideal}, \ldots, \Delta_k$, guarantees that all honest parties decide on k logs $\Lambda_1, \ldots, \Lambda_{ideal}, \ldots, \Lambda_k$. This protocol ensures the following properties $\forall \Delta_{j'} \geqslant \Delta_{ideal}$.*

Safety. *If two honest parties deliver logs $\Lambda_{j'}$ and $\Lambda'_{j'}$, then $\Lambda_{j'}$ and $\Lambda'_{j'}$ do not conflict.*

Liveness. *Blocks containing transactions continue to be appended to $\Lambda_{j'}$.*

4.3.2 Total-Order-Broadcast

In Figures 4.2 and 4.3, we present our protocol for dynamically available TOB under the generalized synchrony model. We start by describing the portion of the protocol presented in Figure 4.2. [6]

Figure 4.2 depicts the sub-protocol that is run k times concurrently, where instance π_j is executed with delay Δ_j. Throughout the course of the protocol, a party maintains two

[6] In this section we assume that joining parties adopt the exact state of some honest party that was already awake prior to it joining. We remove this assumption in section 4.4.

important variables: *est* and *move*. The variable *est* is used to indicate the delay that the party believes is Δ_{ideal}. In protocol π_{est}, the party proposes new blocks and votes for new block proposals; it does not vote for new block proposals in any other sub-protocol. If a party sees evidence that Δ_{est} does not hold during the course of the protocol, they set $move := 1$. In addition to these variables, a party mantains for each j a timestamp t_j^*. Upon setting $est = j$, a party sets t_j^* to the current timestamp t_{now}. The purpose of this timestamp is that parties use it to detect whether they are seeing sufficient progress in π_{est}. We discuss in more detail how it is used later.

The sub-protocol π_j proceeds in epochs of length $10\Delta_j$. Each epoch begins with two instances of GA (Definition 96), as in the approach of [65]. To the first instance of GA, GA_0, the party inputs a block: either their lock from that sub-protocol, lock_j, or their highest decided block from the previous sub-protocol, decide_{j-1} if it extends lock_j. The latter option is a key difference in our protocol from that of [65]. The reason why parties input decide_{j-1} if it extends lock_j is for the liveness and prefix properties: a party must continue to grow all of its logs, and its logs constructed with higher delays must be prefixes of those constructed in lower delays. The set $\mathsf{out}_{0,j}$ contains all of the party's outputs from GA_0, meaning that it is a set of tuples of blocks and their corresponding grades. The party takes the block from $\mathsf{out}_{0,j}$ with the greatest height that is output with grade 1 such that no conflicting blocks were output (with any grade) to be its input to the second GA. From the outputs of the second GA, the party determines its candidate block, B_{cand} and updates lock_j. In particular, it sets lock_j equal to the block with the greatest height output with grade 1, and it sets B_{cand} equal to the block with the greatest height (of any grade) that it output. As described earlier, this approach enables the parties to have consistent locks, and for the parties to choose candidate blocks that extend the locks of all honest parties.

The next component of the sub-protocol is an instance of GPE (Definition 97). If a party has $est = j$, they create a new block proposal extending B_{cand} containing all the new transactions in their buffer. Otherwise, they propose B_{cand}. The parties execute GPE with their block proposal, as well as a specification for how they evaluate the function $\mathsf{predicate}$ on blocks within the GPE. Specifically, if $est = j$, the party only requires that the proposed

block extends lock_j. In other words, the party accepts proposals of newly constructed blocks. Otherwise, the block must also have been output by the party from GA_0, meaning that this block was previously constructed and input to GA_0 by some honest party. The purpose of choosing block proposals and evaluating predicate on proposed blocks in this way is that it ensures that only blocks that were already committed in π_{low}, where low is the current value of est of the honest parties, will be committed in $\pi_{j>low}$. This is necessary because a conflict between Λ_{low} and Λ_j for $j > low$ is seen as evidence that Δ_{low} does not hold, and the parties will attempt to move est past low if this happens. If the parties move est past $ideal$, progress is not ensured at the speed of Δ_{ideal}. Furthermore, such a conflict violates the prefix property, and potentially the safety property as well.

Once the GPE outputs, if the party outputs a block with grade 1, it commits the block. The party also updates lock_j if it output a block (with any grade) from GPE. Note that GPE has a uniqueness property, which ensures that there is a single block $B \neq \perp$ that the honest parties may output from a single instance of GPE. Due to the graded delivery property of GPE, this step ensures that if an honest party commits a block, all of the awake honest parties lock on it. If the delay corresponding to the sub-protocol holds, the fact that when one party decides a block all of the awake parties lock on it ensures that honest parties do not have conflicting chains.

Participation criteria. There are also certain instructions that a party executes at any point during the sub-protocol. If a party decides a block which causes Λ_j to be longer than $\Lambda_{j'<j}$, it commits all the blocks that it has committed in π_j in $\pi_{j'}$ (and updates its locks accordingly). Note that this may cause the party to see a fork in the chain constructed in a sub-protocol, which it uses as evidence that the delay corresponding to that sub-protocol does not hold. If a party meets any of the criteria for moving up, i.e. it sees evidence that Δ_j does not hold, it sets $move := 1$ and ceases participation in π_j.

We now describe these criteria in further detail. Criterion 1 captures that once all honest parties have $est = j$ for the same value of j, if Δ_j holds, the parties should commit κ new blocks every 2κ epochs. As we describe later, it takes $3\Delta_k$ time once one party sets $est = j$ for the rest of the parties to have $est = j$. For this reason, a party expects progress in π_{est}

from time $t^*_{est} + 3\Delta_k$ onward. Criterion 2 captures the fact that a block committed in π_j should be committed in π_{j+1} in a timely manner. This captures the liveness property: that $\Lambda_j \forall j \geqslant \mathit{ideal}$ should continue to grow throughout the duration of the protocol. This property also ensures a more nuanced property. It is possible that Δ_j does not hold, and because of this only a single honest party sees progress in π_j. As we later show, it is necessary that *all* of the awake honest parties have $\mathit{move} := 1$ in order for them to increment est. Due to criterion 2, if a party stops the network from converging on Δ_{ideal}, this is because a party is seeing progress at an *even lower delay*, and this progress is reflected in all the higher logs as well. It ensures that if the parties don't eventually converge on the lowest delay that holds, it is because progress is made at an even faster rate than it would have been if they did converge on Δ_{ideal}. Finally, criterion 3 captures that seeing a conflict between Λ_j and $\Lambda_{j'>j}$ indicates that Δ_j does not hold. The protocol ensures that only blocks committed in lower sub-protocols may be committed or locked on in higher sub-protocols, so a violation of this property is grounds to set $\mathit{move} := 1$.

The π_{beacon} protocol. Next, we describe the protocol π_{beacon} in Figure 4.3. This protocol, executed with a delay of Δ_k, dictates when a party locally decides to increase its value of est. The protocol consists only of GAs executed at intervals of $3\Delta_k$. To each GA, a party inputs the sum of its value of est and its value of move. Within GA, the numbers 1 through k form a chain. That is, 2 extends 1; 3 extends 2; k extends $k-1$; and so on. From the outputs, out, of GA, the party takes the highest value output from GA, v, and the highest grade with which v was output, g. If $g = 1$, and $v > \mathit{est}$, the party ceases participation in π_{est}, sets $\mathit{est} = v$, resets $\mathit{move} := 0$, and sets $t^*_{\mathit{est}} := t_{\mathit{now}}$, the current timestamp. Otherwise, the party only sets $\mathit{move} := v - \mathit{est}$. The graded delivery property of GA ensures that once one honest party sets $\mathit{est} = j$, all of the honest parties have $\mathit{est} = j$ within $3\Delta_k$ time. This property is crucial for our protocol, because progress in π_{est} is ensured only once *all* awake honest parties have est equal to that value. Since a party knows that after it has updated its value of est, all of the other honest parties will do so within $3\Delta_k$ time, it can expect to see progress as soon as that grace period has passed, as reflected in criterion 1 for moving up.

4.3.3 Protocol Analysis

In this section, we prove that the protocol implements dynamically available Total-Order Broadcast under generalized synchrony. In these proofs, we assume that when parties join the protocol, and changes their status to *awake* at time t, they assume the state of another honest party that was awake prior to and including time t. In Section 4.4, we show how to do away with this assumption with our joining protocol. We start by proving that the protocol satisfies a property called *conditional safety*. Conditional safety is safety within a sub-protocol π_j (no two honest parties decide different blocks at the same height) conditioned on two points: (1) Δ_j holds and (2) no honest party has $est > j$. Note that the first condition is equivalent to saying that $j \geqslant ideal$, and we use the two notions interchangeably.

Definition 99 (Conditional Safety). *If Δ_j holds and no honest party has $est > j$, then if an honest party decides block B in protocol π_j, no honest party decides a block B' conflicting with B in π_j.*

We start our protocol analysis with a proof of conditional safety because the safety and prefix properties of our TOB protocol requires that no honest party has incremented *est* past *ideal*. This is because honest parties propose new blocks in the GPE of π_{est} and evaluate $\mathsf{predicate}(B) = \mathsf{true}$ for new blocks in π_{est}. As a result, if some honest party has $est > ideal$ the protocol does not prohibit a conflict between Λ_j for $j > ideal$ and Λ_{ideal}. One of the conditions by which a party decides that they wish to move to a higher sub-protocol is if they see a fork in π_{est} or if the lock in a higher sub-protocol conflicts with the chain they are constructing in π_{est} (condition 3). If conditional safety does not hold, then honest parties may increment *est* past *ideal*. If an honest party has $est > ideal$, then the protocol does not guarantee that blocks are decided at a latency of $O(\Delta_{ideal})$. Since the safety and prefix properties of the protocol and the fact that the property that no honest party increments *est* past *ideal* are interdependent, we start by proving a weaker safety property, conditional safety. The proof of conditional safety serves as a stepping stone to proving that no party increments *est* past *ideal*, which enables us to prove the that the protocol satisfies the properties of TOB under generalized synchrony and that progress is made at the speed of Δ_{ideal}.

We start with two helpful lemmas that are stepping stones to proving that conditional safety holds. In Lemma 100, we show that blocks are decided directly in lower sub-protocols before they are locked on (and as a consequence, also decided), in higher sub-protocols. This is an important step in proving that progress occurs at the speed of Δ_{ideal} and that the safety and prefix properties are preserved. If this property does not hold, Λ_j and $\Lambda_{j'}$ for $j, j' \geqslant ideal$ could conflict, violating these properties.

Lemma 100. *If $j \geqslant ideal$, and all honest parties have $est \leqslant j$, then for all $j' > j$, $\mathsf{lock}_{j'}$ held by any honest party was decided by an honest party in π_j.*

Proof. We prove this lemma by induction on j, demonstrating that the statement holds for $j' = j + 1$. It is evident that the statement holds at the beginning of the protocol. Now, assuming it holds at the start of epoch $e_{j'} \geqslant 0$ of $\pi_{j'}$, we aim to prove that it continues to hold at the start of epoch $e_{j'} + 1$.

At the start of epoch $e_{j'}$, honest parties input to $\mathsf{GA}_{0,j'}$ the highest block they decided in π_j (if it extends $\mathsf{lock}_{j'}$) or $\mathsf{lock}_{j'}$. All honest parties input a block that was decided by an honest party in π_j. This follows directly from the inductive assumption. The honest parties then update $\mathsf{lock}_{j'}$ to the highest grade 1 output from $\mathsf{GA}_{1,j'}$ (line 15). The lemma holds for these lock updates because of the integrity property of GA and the fact that honest parties chose their inputs to $\mathsf{GA}_{1,j'}$ from their outputs from $\mathsf{GA}_{0,j'}$.

A necessary condition for honest parties (all having $est < j'$ by the lemma statement) to evaluate predicate to true on a block in GPE of $\pi_{j'}$ is that they output it (or a block extending it) with any grade from $\mathsf{GA}_{0,j'}$. This implies that if the leader's proposal succeeds, i.e., some honest parties output a non-$\perp$ block B from GPE, then some honest party decided the block B in π_j (by the integrity property of GA). The lemma therefore follows for lock updates after GPE outputs (line 25).

We have thus proven the statement for $j' = j + 1$. The the lemma therefore holds for every two consecutive pairs of sub-protocols where both are at least j.

Next, we address general $j \geqslant ideal$ and $j' > j$. To do so, we will use the fact that we proved the statement for $j' = j + 1$ already. If an honest party sets $\mathsf{lock}_{j'} = B$ in $\pi_{j'}$ at time t,

then some honest party decided it in $\pi_{j'-1}$ prior to time t, which follows from the fact that the lemma holds for $j' = j + 1$. Since an honest party decided B in $\pi_{j'-1}$, they must have locked on it as well (line 25). It follows from the fact that the lemma holds for $j' = j + 1$ that some honest party must have decided B in $\pi_{j'-2}$ prior to this. Using this line of reasoning, we see that the lemma holds for general $j \geqslant \mathit{ideal}$ and $j' > j$. □

Lemma 101. *If an honest party decides block B in epoch e_j of π_j at time t, Δ_j holds, and all honest parties have est $\leqslant j$, then all honest parties awake at any time $t' \geqslant t$ have lock_j extending B from time t' onwards.*

Proof. Note that all honest parties awake at time t update lock_j to B in epoch e_j due to graded delivery of GPE (line 25). We therefore only need to prove that no honest party locks on a block that doesn't extend B in any epoch after e_j.

We use a proof by induction to show that the lemma holds for all epochs after epoch e_j. We already showed that the lemma holds at the end of epoch e_j, as a base case. Assume that it holds at the end of some epoch e'_j. We show that it holds at the end of epoch $e'_j + 1$.

In epoch $e'_j + 1$ of π_j, all awake honest parties input to $\mathsf{GA}_{0,j}$ their lock_j or their highest decided block from π_{j-1} if it extends lock_j. In either case, they input a block extending B. It follows from integrity and validity that all awake honest parties input a block extending B to $\mathsf{GA}_{1,j}$. By integrity, no block output by an honest party from $\mathsf{GA}_{0,j}$ or $\mathsf{GA}_{1,j}$ may conflict with B, and by validity, the lock of all awake honest parties on line 15 extends B. By validity of GPE, if an honest party updates their lock due to outputting a block from GPE, then the block must extend B. It follows that all honest parties that are awake at the end of epoch $e'_j + 1$ have lock_j extending B. □

We are now ready to prove that the protocol satisfies conditional safety.

Lemma 102 (Conditional Safety)**.** *If no honest party has est $> j$ and Δ_j holds, then if an honest party decides block B in protocol π_j, no honest party decides a block B' conflicting with B in π_j.*

Proof. Suppose an honest party h decides block B in protocol π_j in epoch e_j. We need to show that an honest party h' cannot decide on a conflicting block B' in π_j at any time.

Assume that h' decides block B' conflicting with B in epoch e'_j; we will show that this implies a contradiction.

First, note that it cannot be the case that $e'_j = e_j$. This is because of the consistency property of the execution of GPE in epoch e_j. Without loss of generality, let us assume $e_j < e'_j$. Also, since we have $est \leqslant j$, by Lemma 101 all honest parties awake at any time $t' \geqslant t$ have lock_j extending B from time t' onwards. A necessary condition for parties to decide on a leader's proposal in GPE (i.e., evaluate $\mathsf{predicate} = \mathit{true}$ for the proposal) is if it extends their current lock. Thus, since all honest parties have lock_j extending B, no party would decide on a block conflicting with it (by the validity property of GPE). Thus, h' will not decide on B' in any epoch e'. □

Lemma 103. *If honest party p_i sets $est = j$ at time t, all honest parties awake at any time $t' \geqslant t + 3\Delta_k$ have $est \geqslant j$ by time t'.*

Proof. By graded delivery, all honest parties awake at time t output $(j, *)$ from GA at time t. The parties that output $(j, 0)$ set $move := j' - est$, where $j' \geqslant j$ (since they may also output a value higher than j). All honest parties that input to the subsequent GA do so with input at least j. By validity of GA, every awake honest party outputs $(j'', 1)$ for some value $j'' \geqslant j$ from the next GA and updates their value of est accordingly. The parties output from this GA at time $t + 3\Delta_k$, thus proving the lemma. □

Lemma 104. *If Δ_j holds and no honest party has $est > j$, then B_{cand} of every honest party during epoch e_j awake at time $(10e_j + 6)\Delta_j$ extends lock_j of all honest parties awake at that time during epoch e_j from time $(10e_j + 6)\Delta_j$ onward, and a block extending B_{cand} was output by all honest parties awake at any point during epoch e_j from time $(10e_j + 3)\Delta_j$ onward from $\mathsf{GA}_{0,j}$ of that epoch.*

Proof. Because honest parties input to $\mathsf{GA}_{1,j}$ their highest grade 1 output from $\mathsf{GA}_{0,j}$ such that there were no conflicting outputs, it must be the case that none of the outputs of $\mathsf{GA}_{1,j}$ of honest parties conflict during epoch e_j. This follows from the graded delivery and integrity properties of GA. Parties set their lock_j to be the highest grade 1 output from $\mathsf{GA}_{1,j}$. The graded delivery property of GA ensures that all honest parties output the lock_j of all other

honest parties, at least with grade 0. Moreover, B_{cand} is chosen to be the block with the greatest height from the output of $\mathsf{GA}_{0,j}$. Since, the outputs of $\mathsf{GA}_{0,j}$ do not conflict, B_{cand} set by all honest parties awake at time $(10e_j + 6)\Delta_j$ must either be equal to lock_j or a block that extends lock_j for all honest parties that are awake at that time. This proves the first part of the lemma.

To prove that B_{cand} of all honest parties at time $(10e_j + 6)\Delta_j$ was output from $\mathsf{GA}_{0,j}$ of the same epoch by all honest parties that were awake at time $(10e_j + 3)\Delta_j$ onwards, recall that parties choose as B_{cand} their highest output of any grade from $\mathsf{GA}_{1,j}$. The integrity property implies that some honest party must have input a block extending B_{cand} to $\mathsf{GA}_{1,j}$ at time $(10e_j + 3)\Delta_j$. Honest parties choose B_{cand} their highest grade 1 output from $\mathsf{GA}_{0,j}$ (with no conflicting outputs). The graded delivery property ensures that if an honest party outputs a block with grade 1, all honest parties output it with at least grade 0.

□

Lemma 105. *If Δ_j holds, no honest party has est $> j$, $j < k$, and all honest parties awake at time t have decided B (or a block extending it) in π_j by time t, then all honest parties awake at any point at or after t have lock_{j+1} equal to a block extending B for all epochs of π_{j+1} beginning at or after time t.*

Proof. By conditional safety (Lemma 102), no honest party decides a block conflicting with B in π_j. By conditional safety and Lemma 100, it follows that no honest party ever has lock_{j+1} conflicting with B. From the way that parties choose their inputs to $\mathsf{GA}_{0,j+1}$ on lines 9- 10, it follows that all awake honest parties input to $\mathsf{GA}_{0,j+1}$ of π_{j+1} B or a block extending it for all epochs beginning at or after time t.

In every epoch of π_{j+1} starting after time t, by the validity and integrity properties of GA, no honest party outputs a block conflicting with B from $\mathsf{GA}_{0,j+1}$ and all honest parties awake at the time of outputting, output B or a block extending it with grade 1. As a result, all awake honest parties input a block extending B to $\mathsf{GA}_{1,j+1}$. The lemma therefore follows for lock updates prior to GPE due to the integrity and validity properties of GA. Because all awake honest parties are locked on blocks extending B, and by the way honest parties evaluate $\mathsf{predicate}()$, the lemma follows for lock updates after GPE due to integrity. □

In the following lemma, we show that for every 2κ epoch period where all honest parties awake at any point during that time period have $est = j$ for some $j \geqslant ideal$, there are at least κ epochs during that period with high probability s.t. all honest parties awake by the end of that epoch have decided a new block. In addition, we show that for all $j' > j$, during the time period where all awake honest parties have $est = j$, there is at least 1 epoch out of every κ in which a new block is decided by all honest parties awake by the end of that epoch with high probability. Later, we use this lemma to show that once there is a time t s.t. all honest parties awake at any point at or after t have $est = ideal$, they continuously decide sufficiently many new blocks in π_{ideal} which are also confirmed in higher sub-protocols.

Lemma 106. *If there are 2κ consecutive epochs in π_j where all honest parties awake at any point during that time period have $est = j$ and Δ_j holds, then during at least κ of those epochs all honest parties awake by the end of that epoch decide B, where B is the input of an honest party, with probability $1 - e^{-2\kappa\epsilon^2}$. In addition, during the time in which all awake honest parties have $est = j$, $\forall j' > j$ there is at least one epoch out of every κ in which all of the honest parties awake by the end of that epoch output B, where B is the input of an honest party, with probability $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$.*

Proof. We start by proving the first statement. By Lemma 104, and line 19, in each such epoch of π_j, the inputs of all honest parties to GPE satisfy *predicate* for all honest parties that are awake at any point during the GPE. As such, in expectation, a block that is the input of an honest party to GPE is decided every $\frac{1}{\frac{1}{2}+\epsilon}$ epochs in expectation by the validity property of GPE by all honest parties awake by the end of that epoch. With the application of a Chernoff Bound, it follows that with probability $1 - e^{-2\kappa\epsilon^2}$, this occurs at least κ epochs out of every 2κ.

We now address the second statement in the lemma. Again, by Lemma 104, the inputs of all awake honest parties to GPE satisfy *predicate* for all honest parties awake during GPE in every epoch where all honest parties have $est = j$. The remainder of the lemma therefore follows from a simple Chernoff Bound. □

In the next lemma, we show that once one honest party decides a block in $\pi_{j \geqslant ideal}$, all honest parties that are awake κ epochs later will have decided it as well with high probability.

Lemma 107. *If an honest party p_i decides block B directly in epoch e_j of π_j, Δ_j holds, and all honest parties have $est < j$ then all honest parties that are awake at the end of epoch $e_j + \kappa$ will have decided B in π_j with probability $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$.*

Proof. Because p_i decided B directly in π_j in epoch e_j, it follows from Lemma 101 that all honest parties awake at time t' after the end of epoch e_j have lock_j equal to a block extending B from time t' onward. By conditional safety (Lemma 102), no honest party decides a block conflicting with B in π_j. Furthermore, by Lemma 104 and the fact that all honest parties have $est < j$, in all subsequent epochs the blocks proposed by honest parties satisfy *predicate* for all awake honest parties (and therefore extend B).

From the validity property of GPE, it follows that with probability $\frac{1}{2} + \epsilon$ all awake honest parties output $(B', 1)$ from GPE (and therefore decide it) where B' is the input of an honest party and therefore extending B in each such epoch. It follows with the application of a Chernoff Bound that this occurs within κ epochs with probability $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$. □

Lemma 108. *If an honest party sets $est := j$ at time t and no honest party ever has $est + move > j$, then all honest parties awake at time $t' \geqslant t + 3\Delta_k$ onward have $est = j$ by time t'.*

Proof. This follows from the graded delivery and integrity properties of GA . □

We are now ready to prove that with overwhelming probability, no honest party ever sets *est* > *ideal* for the duration of the protocol, which is crucial to ensuring that progress is made at the speed of $O(\Delta_{ideal})$.

Lemma 109. *No honest party ever sets $est > ideal$ except for probability $1 - \mathsf{negl}(\kappa)$.*

Proof. From the protocol in Figure 4.3, an honest party must output a value from $\mathsf{GA_B}$ that is greater than *ideal* in order to set *est* past *ideal*. By the integrity property of GA, this implies that some honest party input a value $j > ideal$ to GA. To do so, that party must already have $est > ideal$ or $est \leqslant ideal$ and $move \geqslant 1$.

Now, we show a proof by contradiction. Let p be the first party to set $est > ideal$ (if multiple did at the same time, choose one arbitrarily without loss of generality), and let t be the time at which they did so. This implies two possibilities prior to time t due to the

integrity property of GA: (1) some honest party p' had $est = ideal$ and set $move := 1$ due to meeting one of the criteria for moving up or (2) some honest party had $est \leqslant ideal$ and $move > ideal - est$ due to outputting a value $v > ideal$ from GA_B in π_{beacon}.

We start by addressing the first case, and check if it is possible for some party p' to have had $est = ideal$ and set $move := 1$ due to meeting one of the criteria for moving up. Without loss of generality, let p' be the first party to have $est = ideal$ and $move := 1$. We go through each possible criteria for moving up to check if it's possible for p' to meet one of them at some time t^* prior to time t, assuming that none of the other criteria are met prior to time t^*. If none of the criteria are met in the absence of any of the other criteria already having been met, it follows that possibility (1) could not have happened.

We start with criterion 1 for moving up, and let $t' \leqslant t^* < t$ be the time at which p' set $est = ideal$. All honest parties awake at time $t' + 3\Delta_k$ onwards must have had $est = ideal$ by time $t' + 3\Delta_k$ by Lemma 108. By Lemma 106 there are κ epochs out of every 2κ epochs from time $t' + 3\Delta_k$ onwards in which all honest parties that are awake by the end of that epoch decide a new block proposed by an honest leader with probability $1 - e^{-2\kappa\epsilon^2}$, and this holds for every 2κ epoch period after that (as long as no honest party increments est past $ideal$ for any other reason).

Next, we address criterion 2. Again, let $t' \leqslant t^* < t$ be the time at which p' set $est = ideal$. All honest parties awake at time $t' + 3\Delta_k$ onward must have had $est = ideal$ by time $t' + 3\Delta_k$ by Lemma 108 (prior to which this criterion could not have been met for p'). By Lemma 107, when an honest party decides a block in π_j for $j \geqslant ideal$ and all awake honest parties have $est = ideal$, all honest parties awake by the end of the subsequent κ epochs will have decided that block with probability $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$. By Lemma 105, once all honest parties awake at a certain time t decide a block in $\pi_{j \geqslant ideal}$, all honest parties awake at that point onward are locked on that block in π_{j+1} for all subsequent epochs beginning at time t onward. By Lemma 104 and Lemma 106, all honest parties awake at the end of the κ subsequent epochs of π_{j+1} will have decided that block with probability $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$. It follows from the fact that $\Delta_j = 2\Delta_{j-1}\ \forall 1 < j \leqslant k$ that any block decided in protocol $\pi_{j'}$ for $ideal \leqslant j < k$ all honest parties awake at the end of the subsequent 2κ epochs of $\pi_{j'+1}$ will

have decided that block with high probability, in which case the criterion cannot be met in the absence of any of the other criterion having been met.

Finally, we address criterion 3. We start by checking whether the first part of the statement is possible: $\mathsf{lock}_{j'}$ conflicts with the chain constructed in π_j for some $j \geqslant ideal$ and $j' > j$. Conditional safety (Lemma 102) applies for π_j since no honest party has $est > ideal$, and Lemma 100 ensures that $\mathsf{lock}_{j'}$ was decided by an honest party in π_j. It follows that this part of the criterion cannot be met. The second part of the statement, that there is a fork in the chain constructed in π_j for $j \geqslant ideal$, also cannot be met due to conditional safety.

It therefore follows that option (1) could not have happened.

Next, we address (2): the possibility that p' had $est \leqslant ideal$ and $move > ideal - est$ due to outputting a value $v > ideal$ from GA in π_{beacon} prior to time t. For this to happen, p' must have output $v \geqslant ideal + 1$ from an instance of GA in π_{beacon} prior to time t. By the integrity property of GA, some honest party p'' must have input $v' \geqslant v$ to GA prior to time t. Without loss of generality, assume that p'' is the first party to have done so prior to time t. As already addressed when exploring possibility (1), it is not possible for p'' to have done so due to having $est = ideal$ and $move = 1$ from meeting the criteria for moving up with high probability. By the definition of p'', possibility 2 also cannot be because it would imply that a party input $v'' \geqslant v$ to GA prior to p'' doing so. We have arrived at a contradiction. □

In the following two lemmas, we prove that the protocol satisfies the properties of TOB.

Lemma 110 (Liveness)**.** *Blocks containing transactions continue to be appended to* $\Lambda_j \forall j \geqslant ideal$.

Proof. If all awake honest parties have $est = j$ for $j \geqslant ideal$, the lemma follows from Lemma 106 and the fact that when an honest party decides a block in a sub-protocol π_j, they decide it to all $\Lambda_{j'}$ for $j' < j$. If all honest parties have $est < ideal$, this implies that some honest party p_i never meets any of the criteria for moving up past some sub-protocol π_j for $j < ideal$. This implies, by criterion 1 for moving up, that p_i sees κ new blocks decided every 2κ epochs to Λ_j. Furthermore, by criterion 2, every block decided by p_i to $\Lambda_{j'}$ is decided within the following 2κ epochs of $\pi_{j'+1}$ to $\Lambda_{j'+1} \forall j \leqslant j' < k$. For p_i, the fact that they don't meet criterion 3 for moving up implies that $\Lambda_{j'}$ and $\Lambda_{j''}$ do not conflict for any $j' \geqslant j$ and

$j'' \geqslant j$. This follows from the fact that if they decide a block in any sub-protocol, they also update their lock for that sub-protocol so that it extends that block. It follows that every block that p_i decides to Λ_j, it also eventually decides to $\Lambda_{ideal,\ldots,k}$. □

Lemma 111 (Safety)**.** *If two honest parties deliver logs $\Lambda_{j'}$ and $\Lambda'_{j'}$, then $\Lambda_{j'}$ and $\Lambda'_{j'}$ do not conflict $\forall \Delta_{j'} \geqslant \Delta_{ideal}$.*

Proof. The lemma follows from conditional safety (Lemma 102) and the fact that no honest party ever has $est > ideal$ (Lemma 109). □

Finally, we prove that blocks are decided at the desired rate in expectation.

Lemma 112. *Blocks are eventually decided at a rate of 1 block per $O(\Delta_{ideal})$ time in expectation.*

Proof. To prove the lemma statement, we show that in all π_j for $ideal \leqslant j \leqslant k$, $2^{j-ideal}\kappa$ blocks are decided every 2κ epochs. Because $\Delta_j = \frac{\Delta_{j+1}}{2}\ \forall j < k$, this implies the lemma statement.

First, we prove the claim that new blocks are decided at a rate of κ blocks every 2κ epochs in π_{ideal}. By Lemma 109, we know that all honest parties have $est \leqslant ideal$ for the duration of the protocol. If there eventually comes a time that all awake honest parties have $est = ideal$, the claim follows from Lemma 106. If this does not eventually happen, validity of GA implies that for every iteration of π_{beacon}, some honest party inputs $est + move < ideal$. Eventually, by validity of GPE, there must be some $j < ideal$ such that some honest party (note that it need not be the same one each time) inputs j to each iteration of π_{beacon} for the remainder of the protocol. Otherwise, at some point, all the awake honest parties *would* reach $est = ideal$.

Let $t^*_{1,j}$ be the earliest time at which an iteration of π_{beacon} output $(j, 1)$ for an honest party. By Lemma 103, all honest parties awake at time $t^*_{2,j} = t^*_{1,j} + 3\Delta_k$ must have output $(j, 1)$ by that time. Note that no honest party could output $(j', 1)$ for $j' > j$ from π_{beacon}, as this would contradict the definition of j by Lemma 103.

Let t be some time during the course of the protocol s.t. $t > t^*_{2,j}$. Let p be the party that input j to the latest instance of π_{beacon} that completed at or prior to time t with $move = 0$.

This implies that p never met any of the criteria for moving up since time $t_j^* \leqslant t_{2,j}^*$ (during which time it may have been awake or based on the state it acquired when joining the protocol). That is, p decides κ new blocks for every 2κ epochs of π_j from time $t_j^* + 3\Delta_k$ onward. Furthermore, every block it decides to $\Lambda_{j'}$, it decides within the subsequent 2κ epochs of $\pi_{j'+1}$ to $\Lambda_{j'+1}$. By Lemma 107, and the fact that $\Delta_{j+1} = 2\Delta_j \forall j < k$, it follows that in all π_j for $ideal \leqslant j \leqslant k$, all honest parties awake at the end of every 2κ epoch period will have decided κ new blocks with probability $1 - \mathsf{negl}(\kappa)$. □

Corollary 113. *If there exists some time t s.t. all honest parties awake at time t have est = ideal, then beginning at time t onward, all honest parties awake at the end of every 2κ epoch period will have decided κ new blocks and any block decided by an honest party in π_j for $ideal \leqslant j < k$ is decided by all honest parties in π_{j+1} that are awake at the end of the subsequent 2κ epochs of π_{j+1} with high probability.*

4.4 Joining Parties

In this section, we present our joining protocol which enables us to remove the assumption that joining parties adopt the state of an already-awake honest party.

As described in the introduction, joining parties need to learn a value of *est* and *move* to move so that joining parties don't hinder the liveness of the protocol. In addition, they must learn ad log Λ_j and a lock lock_j for each sub-protocol π_j. Using this information, they should be able to check that a sufficient amount of progress has been made in π_j for it to be the case that $est = ideal$, and that none of the other criteria for incrementing *est* have been met. For this purpose, the joining party needs two additional pieces of information: t_{est}^* and $\mathsf{backlog}_{est}$. Here, t_{est}^* is the time at which the honest parties that were awake at the time of updating *est* to this value did so. $\mathsf{backlog}_{est}$ is (an approximation of) the log Λ_{est} that those parties had prior to updating *est* to this value. Using this information, the joining party can detect if it is possible that κ blocks were committed every 2κ epochs in π_{est} since all of the awake honest parties adopted this value of *est*, and set $move := 1$ if not.

We now describe the joining protocol in detail and how the joining party captures all of this information. We show how it ensures that a joining party p_{join} obtains the following

information so that it may begin participation in the protocol without harming the safety or stalling liveness.

1. for each sub-protocol π_j, a log Λ_j and a lock lock_j that captures the current state of the awake honest parties
2. a value of *est* and *move* to begin with
3. if it should set $move := 1$ prior to joining the protocol due to any of the criteria for moving up having already been met.

Figure 4.4 presents the joining protocol executed by all awake parties and the joining party, p_{join}. On lines 4, parties update their backlogs each time they update their value of *est*. They update their backlog to the head of any of their chains $\Lambda j \geqslant est$ such that the block does not conflict with any chains $\Lambda_{j'>j}$. Lines 5- 9 depict a series of graded agreements (GA) running continuously at $3\Delta_k$ intervals with a delay of Δ_k. For each sub-protocol π_j— $\mathsf{GA}_{high,j}$ and $\mathsf{GA}_{lock,j}$— are executed concurrently across all sub-protocols $\pi_1, \ldots, \pi_k$. While only awake parties participate in these GA instances and send messages, p_{join} records their outputs as specified in line 7. The outputs of these GA's are later used by p_{join} to obtain item 1. In addition, the parties input to $\mathsf{GA}_{backlog,j}$ $backlog_j$ if $j \leqslant est$, and $height(\mathsf{decide}_{j'})$ otherwise, where $j' \geqslant j$ and $\mathsf{decide}_{j'}$ does not conflict with $\Lambda_{j''\forall j'' \geqslant j'}$. Note that we assume for $\mathsf{GA}_{backlog,j}$ that the set of non-negative integers form a chain, where $i+1$ extends i. This information is later used by joining parties to determine item 3.

Lines 10- 19 depict actions done by p_{join}. p_{join} starts by setting its status to *recovering*. In addition, it begins storing all messages that it receives and the time at which they were received. The purpose of this is that later during the joining protocol, after adopting its initial state based on $\mathsf{GA}_{status,high,j}$ and $\mathsf{GA}_{status,lock,j}$, p_{join} will add any additional updates to its state by simulating the protocol using the stored messages.

On line 11, p_{join} sends to all of the awake parties a message $\langle \mathsf{recovering}, p_{join}, ts \rangle_{join}$. In this message, p_{join} includes a timestamp, ts, of the start of the next full iteration of π_{beacon}. The inclusion of ts is necessary because at the completion of the instance of π_{beacon} beginning at ts, parties must send to p_{join} information relevant to the value of *est* that they ended the instance with. Note that within the instance of π_{beacon}, p_{join} sets its value of *est* and *move*,

obtaining item 2. On lines 14- 15, p_{join} detects evidence of a fork in the logs constructed by any of the honest parties for $\pi_{j \geqslant est}$, and if so, uses this as evidence that *est* does not hold and sets $move := 1$ accordingly, enabling it to detect item 3.

On lines 16- 19, p_{join} uses the information that it learns from $\mathsf{GA}_{high,j}$ and $\mathsf{GA}_{lock,j}$ to adopt its log and lock for each sub-protocol (for learning item 1). It uses the highest block that it outputs from each of these GAs. The integrity and validity properties of GA ensure that the conditional safety property of all $\pi_{j \geqslant ideal}$ continues to hold with the status adopted by p_{join}. p_{join}'s log corresponding to such sub-protocols will not conflict with the logs of any other honest parties. In addition, each of p_{join}'s logs for each such sub-protocol are a prefix of the log of some honest party awake at time ts, and the log of some honest party awake at time ts is a prefix of p_{join}'s log for each sub-protocol. These properties ensure that properties related to the safety of the main protocol remain intact. Furthermore, if all of the awake parties have decided a block in a given sub-protocol π_j, p_{join} will also have decided that block. The purpose of $\mathsf{GA}_{lock,j}$ is for p_{join} to learn which lock, lock_j, to adopt for each sub-protocol π_j. The validity property ensures that p_{join} chooses a lock that is at least as high as the highest common ancestor of the lock of all of the honest parties at the time at which the GA instances start. Since the main protocol has the property that if an honest party decides a block in sub-protocol π_j, all awake honest parties lock on it (assuming that Δ_j holds), this ensures that p_{join} does not disrupt the safety of any sub-protocol (or the larger protocol), as its lock incorporates any block decided by any honest party.

Lines 20-22 illustrate the actions of awake parties upon receiving the message $\langle \mathsf{recovering}, p_{join}, ts \rangle_{join}$ from p_{join}. After completing one full iteration of π_{beacon} starting at time ts, the awake party p_i sends p_{join} a message $\langle \mathsf{status}, est, t^*_{est} \rangle_i$, which includes (1) its current value of *est* and (2) t^*_{est}, the time of its last *est* update.

Finally, in lines 23- 26, p_{join} uses the information it learned from the awake parties to determine whether it should set $move := 1$ prior to setting its status to *awake*, for item 3. It starts by calculating t^*_{est} based on the status messages it has received. It compiles the values of t^*_{est} reported by each of the responding parties that reported having the same value of *est*, adding t_{now} (the current timestamp) for each party that sent a different value of *est*.

It sets t^*_{est} to the median of these values. The purpose of calculating t^*_{est} is that p_{join} must verify that sufficient progress has been made since the honest parties originally adopted this value of est; otherwise we run into the liveness problem described in the introduction to this section. Note also that if an honest party has a greater value of est than p_{join}'s, these calculations are redundant, because the graded delivery and validity properties of GA ensure that p_{join} will update its values of est and $move$ with the next iteration of π_{beacon}. If an honest party has a lower value of est, the current timestamp is a correct lower bound for the time at which that party will update est to this value. This method of calculating t^*_{est} ensures that (when this calculation is necessary rather than redundant), t^*_{est} is no lower than the time at which the first honest party to adopt this value of est did so, and is at most $3\Delta_k$ higher than that. Since there is up to a $3\Delta_k$ latency in the time between two honest parties that are awake adopting the same value of est (by Lemma 103), p_{join} adopts a "valid" value for t^*_{est}.

Next p_{join} performs calculations to detect whether the logs it is adopting for $\pi_{est,\ldots,k}$ are consistent with Δ_{est} holding. That is, it should see the equivalent of κ blocks committed every 2κ epochs in π_{est} from time $t^*_{est} + 3\Delta_k$ onward, and all but the most recent blocks in Λ_j should have been decided in Λ_{j+1} for $est \leqslant j < k$. To do this, p_{join} calculates $min\_blocks_{j \in est,\ldots,k}$, the minimum number of blocks it should have committed to $\Lambda_{j \in est,\ldots,k}$ when creating its logs on line 19 if criteria 1 and 2 for moving up have *not* been met. min_blocks_{est} is calculated as κ blocks for every 2κ epoch period after $t^*_{est} + 3\Delta_k$. p_{join} sets $min\_blocks_j := \max(min\_blocks_{j-1} - 2^{j-est}\kappa, 0)$. This is an approximation to reflect criterion 2: a block decided in π_j should be decided in π_{j+1} within the subsequent 2κ epochs. On line 31, if fewer than $min\_blocks_j + backlog_{est}$ blocks were decided to any log on line 19, p_{join} sets $move := 1$.

On line 36, p_{join} sets a commit time $t_{B,j}$ for blocks B committed to Λ_j in excess of $min\_blocks_{j+1} + backlog_{est}$. This is because it has not yet committed these blocks to Λ_{j+1} and it needs to ensure that these blocks are committed in π_{j+1} within subsequent 2κ epochs after $t_{B,j}$, as per criterion 2. While $t_{B,j}$ is only guaranteed to be lower bounded by the actual time at which an awake honest party committed B in π_j, it is sufficient for ensuring that

progress is made at the desired rate. This is because up to the end of the last full 2κ epoch period prior to ts, sufficiently many blocks have been decided from p_{join}'s perspective. If p_{join} lets these blocks in excess of min_blocks_j contribute to the next 2κ epoch period (which must contain timestamp ts), even if they were actually decided by an honest party prior to this period, because the previous periods already saw enough progress without these blocks, this is sufficient for driving progress at the desired rate. Furthermore, the blocks that p_{join} decides due to messages received since the time it set its state to *recovering will* contain the actual timestamp of when they were decided by an honest party. deciding a block B implies deciding all blocks that it extends, so p_{join} deciding κ blocks every 2κ epochs in π_{est} and not meeting criteria 2 for moving up is sufficient.

Upon completion of an entire epoch of π_k since beginning the joining protocol, p_{join} updates its entire state (except for updates to *move* due to meeting any criteria for moving up) by simulating the protocol upon the state that it has adopted thus far on line 38. Note that this may include deciding blocks, updating locks, and updating est and *move* (due to π_{beacon}). It is important to note that simulation of the protocol is only to learn what may have happened since p_{join} started the joining protocol; p_{join} does send any messages in the simulation; it just calculates changes to its state based on the saved messages. On line 39, p_{join} starts checking if any of the criteria for moving up are met, from a specific point in time: $t^*_{est} + 3\Delta_k + 20\kappa\Delta_{est}\lfloor\frac{ts-(t^*_{est}+3\Delta_k)}{20\kappa\Delta_{est}}\rfloor$ onward. This time is calculated to begin immediately after the end of the last 2κ epoch period which was covered by min_blocks_{est} (if there was one), and therefore already verified to pass the check. On line 40, p_{join} begins checking criterion 2 for moving up with the blocks that have been decided to a log Λ_j but not to the log of the sub-protocol immediately higher, Λ_{j+1}. On 42, p_{join} begins checking criterion 3. Finally, p_{join} sets its status to *awake*.

The backlog While we cannot ensure that the backlog learned by p_{join} is equal to the largest backlog of any honest party, it is not very far off. Intuitively, the reason is that if an honest party has committed many blocks to Λ_j prior to setting $est = j$, and it continues to believe that $est = ideal$ (meaning it does not set $move := 1$), then the blocks that it commits in π_j, it also commits in all higher sub-protocols, including in π_{ideal}. If one honest

party commits a block in π_{ideal}, all of the awake honest parties commit it within the κ subsequent epochs. Because of the way honest parties set $\mathsf{backlog}_j$ and decide what to input to $\mathsf{GA}_{backlog,j}$, it follows that the backlog learned by a joining party cannot be very far off from the backlog held by any other honest party that is seeing progress in π_j. We formalize this intuition in Appendix 4.6 when analyzing the full protocol.

4.4.1 Recovery Protocol Analysis

We now present the analysis of the recovery protocol along with our analysis of the full TOB protocol. Our approach in this section is that we first show that the state adopted by p_{join} is "safe". That is, for all sub-protocols in which the delay holds, p_{join} adopts a chain that includes the highest decided block at time ts of some honest party that is awake at time ts and that its lock for each such sub-protocol is at least the highest common ancestor of the lock at time ts of all honest parties awake at time ts.

Lemma 114. *Let $\Lambda_{j,ts,join}$ be the chain adopted by p_{join} on line 19 for sub-protocol π_j and $\mathsf{lock}_{j,ts,join}$ be the lock adopted for the same sub-protocol by p_{join} on line 17. Similarly, let $\Lambda_{j,ts,\alpha}$ be Λ_j for party p_α awake at time ts. The following guarantees hold for all $j \geqslant ideal$ if no two awake honest parties have conflicting logs for sub-protocol π_j at time ts:*

1. *$\Lambda_{j,ts,join}$ is a prefix of $\Lambda_{j,ts,\alpha}$ for some honest party p_α awake at time ts, and $\Lambda_{j,ts,\beta}$ of some honest party p_β awake at time ts is a prefix of $\Lambda_{j,ts,join}$.*
2. *$\mathsf{lock}_{j,ts,join}$ extends the highest common ancestor of lock_j for all honest parties awake at time ts.*

Proof. We start by proving the first of the two guarantees. First, we show that $\Lambda_{j,ts,join}$ is a prefix of $\Lambda_{j,ts,\alpha}$ for some honest party p_α awake at time ts. This follows from the integrity property of GA, which applies to $\mathsf{GA}_{status,high,j}$. Next, we show that $\Lambda_{j,ts,\beta}$ of some honest party p_β awake at time ts is a prefix of $\Lambda_{j,ts,join}$. Since no two honest parties have conflicting chains Λ_j for sub-protocol π_j at time ts, the validity property of GA ensures that the highest decided block for some honest party at time ts in sub-protocol π_j must be in $\mathsf{out}_{j,high}$ for p_{join}, from which the statement follows. The second guarantee follows from the validity property of GA. □

From this lemma and the properties proved for the main protocol in the previous section, we derive the following corollaries which are analogous to Lemmas 101, 102, and Lemma 105 for the state adopted by p_{join}.

Corollary 115. *If an honest party directly decides block B in epoch e of π_j, Δ_j holds, and all honest parties have est $\leqslant j$, then p_{join} is locked on a block B′ extending B for all subsequent epochs of π_j from time $ts + 3\Delta_k$ onwards.*

Corollary 116. *If no honest party has est $> j$ and Δ_j holds, then if an honest party decides block B in protocol π_j, p_{join} never decides a block B′ conflicting with B.*

Corollary 117. *If Δ_j holds, no honest party has est $> j$, $j < k$, and all honest parties awake at time t have decided B (or a block extending it) in π_j by time t, then from time max$(t, ts + 3\Delta_k)$ onward, p_{join} has lock_{j+1} equal to a block extending B for all epochs of π_{j+1}.*

In the following lemma we show that the value t^*_{est} adopted by p_{est} is sufficiently close to the value of t^*_{est} held by the first honest party to adopt that value of *est*.

Lemma 118. *Let $t^*_{1,j}$ be the first time at which the GA in π_{beacon} output $(j, 1)$ for an honest party. If no honest party has est $> j$ by time $ts + 3\Delta_k$, then $t^*_{1,j} \leqslant t^*_j \leqslant t^*_{1,j} + 3\Delta_k$, where t^*_j is the value set by p_{join} on line 26.*

Proof. If $t^*_{1,j}$ is at or prior to *ts*, the result follows from the graded delivery property of GA (which ensures that all honest parties that were awake at the time had $est = j$ by time $t^*_{1,j} + 3\Delta_k$) and the fact that p_{join} takes the median of the values it receives in all of the messages, a majority of which must be honest parties. Otherwise, the result follows from the fact that $t^*_{1,k=j}$ must be equal to t^*_j because no honest party has set $est = j$ prior to p_{join} doing so. □

In Appendix 4.6, we prove that the full protocol satisfies the properties of TOB and that blocks are eventually committed at a rate of 1 block per $O(\Delta_{ideal})$ time, in expectation.

4.4.2 Tolerating Backward Simulation

To determine T_b, we start by enumerating the cases in which parties use information from previously executed instances of GA and GPE for each sub-protocol below:

1. As input to GA_0 the parties need the highest committed block from the previous sub-protocol and the lock from the current sub-protocol.
2. As input to GA_1, parties need their output from GA_0.
3. As input to, and for evaluating the predicate within, GPE, parties need their outputs from GA_0 and GA_1.

For item 2, parties use as input to GA_1 their output from GA_0 immediately after it is produced. For item 3, parties use their outputs from GA_0 and GA_1 throughout the instance of GPE (in the same epoch). This information is generated at most $7\Delta_j$ earlier for sub-protocol π_j. For item 1, parties use information that may have been generated epochs ago. While the lock can be generated, at the earliest, in the previous epoch, the highest committed block of the lower sub-protocol could have been generated long ago. However, the joining protocol ensures that when a party sets their status to *awake*, they have this information if necessary. If a party was *awake* at the time of committing their highest committed block in π_{j-1}, they have this information when it's needed in all subsequent epochs of π_j If the party was not *awake*, then they learned the highest committed block of some honest party in π_{j-1} and lock_j from $\mathsf{GA}_{high,j}$ and $\mathsf{GA}_{lock,j}$, respectively. Because they listen for an entire epoch of π_k (and all epochs of the lower sub-protocols that elapse in that time) prior to setting their status to *awake*, if this information changed since the point at which those instances of $\mathsf{GA}_{high,j}$ and $\mathsf{GA}_{lock,j}$ began, the joining party is able to compute the information itself. Since a party doesn't join the protocol in the middle of an epoch, they have the necessary information for participating in GPE because they were awake during the instances of GA_0 and GA_1 in which it was calculated and were able to calculate it themselves.

Since parties wait for the duration of an entire epoch of π_k before setting their status to *awake*, it is necessary that the adversarial fraction grows at a delay that accounts for this. It follows that if $T_b = 20\Delta_k$, that is, the adversarial fraction grows proportionate to the overall participation at a delay of $20\Delta_k$.

4.5 Related Work

In recent years, research on distributed consensus has evolved to address the challenges posed by large-scale systems such as blockchain networks. One of the primary challenges is dynamic participation, where parties in a consensus protocol may go offline and return at any point during execution. This concept was first introduced by Bitcoin [71], which allowed participants to join the network at any time while assuming that a majority of the total computational power (hash rate) remains honest.

Building on this idea, Pass and Shi [72] formally characterized dynamic participation through the Sleepy Model, where protocol execution occurs in discrete rounds, allowing participants to wake up or go offline freely. Their model assumes that in each round, a majority of online nodes are honest. They also proposed a longest-chain consensus protocol tailored for this setting, inspiring further research in the field.

Subsequent works have focused on improving latency and adaptability under dynamic participation. Momose and Ren [68] introduced a Total-Order Broadcast (TOB) protocol with constant latency in the Sleepy Model. Their approach leveraged a Graded Agreement primitive and introduced the time-shifted quorum technique to handle fluctuating participation levels. This technique redefines quorum configurations dynamically to address challenges in forwarding quorum certificates. However, their protocol incurred a relatively high latency of 16Δ, limiting its practicality.

Malkhi, Momose, and Ren [65] improved upon this by proposing a TOB protocol with a best-case latency of 4Δ, using Graded Agreement and a new Graded Proposal Election primitive while maintaining minority corruption tolerance. Their earlier work [64], partially detailed in Appendix A of [65], presented TOB protocols that achieve 3Δ latency with $1/3$ fault tolerance and 2Δ latency with $1/4$ fault tolerance.

In parallel, Gafni and Losa [46] introduced two TOB protocols within the Sleepy Model that also tolerate minority corruption. The first protocol guarantees deterministic safety and probabilistic liveness with constant expected latency, while the second provides both deterministic safety and liveness.

D'Amato *et al.* [27] presented TOB-SDV, a deterministically safe TOB protocol in the

Sleepy Model that tolerates up to 50% adversarial corruption. This protocol improves upon prior work by requiring only a single voting phase per decision in the best case and achieving lower expected latency than existing approaches with the same optimal adversarial resilience.

Beyond deterministically safe protocols, efforts have also been directed toward probabilistically safe dynamically available protocols. While the previously mentioned works focus on ensuring deterministic safety, other research has aimed to design protocols that provide probabilistic safety under dynamic availability. D'Amato *et al.* [26] introduced Goldfish, a dynamically available TOB protocol based on a longest/heaviest chain approach.

Further extending this line of work, D'Amato and Zanolini [25] proposed RLMD-GHOST, a probabilistically safe dynamically available TOB protocol designed for practical deployment. However, a key limitation of dynamically available protocols is their requirement for synchrony [58]. As demonstrated by D'Amato and Zanolini, even short periods of asynchrony in Goldfish can invalidate all prior decisions. To mitigate this, they introduced a mechanism to withstand brief asynchrony periods, though at the cost of slightly reduced adversarial resilience.

D'Amato, Losa, and Zanolini [24] further extended this mechanism to deterministically safe protocols. They demonstrated that by enhancing messages with an expiration period of η rounds and reducing adversarial resilience accordingly, deterministically safe protocols can be made resilient to asynchrony lasting up to $\eta - 1$ rounds.

4.6 Full Protocol Analysis

In the following, we refer to the protocol in Figure 4.2 as the *main protocol* and the protocol in Figure 4.4 as the *joining protocol*.

Lemma 119. *If $j \geqslant$ ideal, and all honest parties have est $\leqslant j$, then for all $j' > j$, $\mathsf{lock}_{j'}$ held by any honest party was decided by an honest party in π_j.*

Proof. There are 4 points at which lock updates occur: in the main protocol, there are those on lines 15 and 25. We refer to these as cases (i) and (ii) respectively. In the joining protocol, there are the lock updates on line 17 and 38. We refer to these as cases (iii) and (iv) respectively.

We prove this lemma by induction on j, demonstrating that the statement holds for $j' = j + 1$.

We start with cases (i), (ii), and (iv), and we use a proof by induction. It is evident that the statement holds at the beginning of the protocol. Now, assuming it holds at the start of epoch $e_{j'} \geqslant 0$ of $\pi_{j'}$, we aim to prove that it continues to hold at the start of epoch $e_{j'} + 1$ (for cases (i), (ii), and (iv)).

First, we address case (i). At the start of epoch $e_{j'}$, honest parties input to $\mathsf{GA}_{0,j'}$ the highest block they decided in π_j (if it extends $\mathsf{lock}_{j'}$) or $\mathsf{lock}_{j'}$. All honest parties input a block that was decided by an honest party in π_j. This follows directly from the inductive assumption. The honest parties then update $\mathsf{lock}_{j'}$ to the highest grade 1 output from $\mathsf{GA}_{1,j'}$ (line 15). The lemma holds for these lock updates because of the integrity property of GA and the fact that honest parties chose their inputs to $\mathsf{GA}_{1,j'}$ from their outputs from $\mathsf{GA}_{0,j'}$.

Next, we address case (ii). A necessary condition for honest parties (all having $est < j'$ by the lemma statement) to evaluate predicate to true on a block in GPE of $\pi_{j'}$ is that they output it (or a block extending it) with any grade from $\mathsf{GA}_{0,j'}$. This implies that if the leader's proposal succeeds, i.e., some honest parties output a non-$\perp$ block B from GPE, then some honest party decided the block B in π_j (by the integrity property of GA). The lemma therefore follows for lock updates after GPE outputs (line 25), case (ii).

Finally, observe that the logic for case (iv) is the same as the logic for cases (i) and (ii); the only difference is that the locking party has status *recovering*, but this does not change the argument. It follows that the lemma statement holds for $j' = j + 1$ and for lock updates due to cases (i), (ii), and (iv.)

Next, we address case (iii). We do not use a proof by induction for this case; all we need to show is that if an honest party locks on a block due to case (iii), some honest party already locked on it due to cases (i), (ii) or (iv) prior to that time; since we showed that the statement holds for lock updates due to cases (i), (ii) and (iv), this proves the statement for case (iii). If an honest party sets lock_j to a block B due to case (iii), it must have output that block from $\mathsf{GA}_{lock,j}$. It follows from the integrity property of GA that some honest party must have input a block extending B to that instance of $\mathsf{GA}_{lock,j}$, and therefore have had

lock_j extending B. By the integrity property of GA, the first party to have set lock_j equal to a block B' extending B couldn't have done so due to case (iii), as it had to have output B' from $\mathsf{GA}_{lock,j}$.

We have thus proven the statement for $j' = j + 1$. The the lemma therefore holds for every two consecutive pairs of sub-protocols where both are at least j.

Next, we address general $j \geqslant ideal$ and $j' > j$. To do so, we will use the fact that we proved the statement for $j' = j + 1$ already. If an honest party sets $\mathsf{lock}_{j'} = B$ in $\pi_{j'}$ at time t, then some honest party decided it in $\pi_{j'-1}$ prior to time t, which follows from the fact that the lemma holds for $j' = j + 1$. Since an honest party decided B in $\pi_{j'-1}$, they must have locked on it as well. This follows directly from the fact that if an honest party decides a block due to line 26 of the main protocol, then they lock on it on line 25 [7]. Otherwise, they must have decided it on line 19 of the joining protocol, in which case it follows from the integrity property of GA, which ensures that some awake honest party must have already decided that block and input it to $\mathsf{GA}_{high,j}$ (implying that the *first* honest party to decide the block must have done so due to line 26 of the main protocol and therefore locked on it on line 25). It follows from the fact that the lemma holds for $j' = j + 1$ that some honest party must have decided B in $\pi_{j'-2}$ prior to this. Using this line of reasoning, we see that the lemma holds for general $j \geqslant ideal$ and $j' > j$. □

We say an honest party decides a block *indirectly* if it decides it on line 19 of the joining protocol; otherwise, we say that it decides it *directly*.

Lemma 120. *If an honest party decides block B directly in epoch e_j of π_j, Δ_j holds, and all honest parties have est $\leqslant j$, then all honest parties awake at any time t' in any epoch $e'_j > e_j$ have lock_j extending B from time t' onward.*

Proof. Note that all honest parties awake or in the recovering state at the time that the GPE of epoch e_j of π_j outputs update lock_j to B due to graded delivery of GPE (line 25). We need to show that all honest parties that are awake at any time t' during an epoch $e'_j > e_j$ also

[7] Note that this could also be when they are simulating the protocol during the joining protocol.

have lock_j equal to a block extending B. We therefore only need to prove that no awake or recovering honest party locks on a block that doesn't extend B in any epoch after e_j.

We use a proof by induction to show that the lemma holds for all epochs after epoch e_j. We already showed that the lemma holds at the end of epoch e_j, as a base case. Assume that it continues to hold up to the end of some epoch $e'_j \geqslant e_j$. We show that it holds up to the end of epoch $e'_j + 1$.

In epoch $e'_j + 1$ of π_j, all honest awake parties input to $\mathsf{GA}_{0,j}$ their lock_j or their highest decided block from π_{j-1} if it extends lock_j. In either case, they input a block extending B. It follows from integrity and validity that all honest awake parties input a block extending B. By integrity, no block output by an honest party (whether they are awake or simulating the protocol) from $\mathsf{GA}_{0,j}$ or $\mathsf{GA}_{1,j}$ may conflict with B, and by validity, the lock of all honest parties set on line 15 extends B. By validity of GPE, if an honest party updates their lock due to outputting a block from GPE, then the block must extend B. Note that by the condition on line 37 of the joining protocol, all honest parties that join the protocol do so only after listening to (and therefore simulating) the protocol for one full epoch of π_k (which must include one full epoch of all $\pi_{j''<k}$), so what we have shown implies that all honest parties in recovery during epoch $e'_j + 1$ set lock_j equal to a block extending B during simulation of the protocol. It follows that all honest parties that are awake at the end of epoch $e'_j + 1$ have lock_j extending B. □

Lemma 121 (Conditional Safety)**.** *If no honest party has est $> j$ and Δ_j holds, then if an honest party decides block B in protocol π_j, no honest party decides a block B' conflicting with B in π_j.*

Proof. Suppose an honest party h decides block B in protocol π_j in epoch e_j. We need to show that an honest party h' cannot decide on a conflicting block B' in π_j at any time, whether directly or indirectly. Assume that h' decides block B' conflicting with B directly in epoch e'_j; we will show that this implies a contradiction.

First, note that it cannot be the case that $e'_j = e_j$. This is because of the consistency property of the execution of GPE in epoch e_j. Without loss of generality, let us assume $e_j < e'_j$. Also, since we have $est \leqslant j$, by Lemma 120 all honest parties awake at any time

$t' \geqslant t$ have lock_j extending B from time t' onwards. A necessary condition for parties to decide on a leader's proposal in GPE (i.e., evaluate predicate $= \mathit{true}$ for the proposal) is if it extends their current lock. Thus, since all awake honest parties have lock_j extending B, no party would directly decide on a block conflicting with it (by the validity property of GPE). Thus, h' will not decide B' directly in any epoch e'.

Next, we show that no honest party h' decides a block B' conflicting with B *indirectly* after epoch e. For this to happen h' must output B' or a block extending it from $\mathsf{GA}_{high,j}$. By integrity of GA, this implies that some awake honest party input to $\mathsf{GA}_{high,j}$ B' or a block extending it. We already showed that no honest party directly decides a block B' conflicting with B. So it must be the case that the honest party that input B' (or a block extending it) to $\mathsf{GA}_{high,j}$ decided it indirectly. W.l.o.g. let h'' be the first honest party to decide a block B' conflicting with B (note that it must be indirectly decided). It must have output it from $\mathsf{GA}_{high,j}$, which implies a contradiction because it means some other honest party must have decided B' before h'' by integrity of GA. □

Lemma 122. *If Δ_j holds and no honest party has est $> j$, then B_{cand} of every honest party during epoch e_j awake at time $(10e_j + 6)\Delta_j$ extends lock_j of all honest parties awake at that time, and a block extending B_{cand} was output by all honest parties awake at any point during epoch e_j from time $(10e_j + 3)\Delta_j$ onward from $\mathsf{GA}_{0,j}$ of that epoch.*

Proof. Because honest parties input to $\mathsf{GA}_{1,j}$ their highest grade 1 output from $\mathsf{GA}_{0,j}$ such that there were no conflicting outputs, it must be the case that none of the outputs of $\mathsf{GA}_{1,j}$ of honest parties conflict during epoch e_j. This follows from the graded delivery and integrity properties of GA. Parties set their lock_j to be the highest grade 1 output from $\mathsf{GA}_{1,j}$. The graded delivery property of GA ensures that all honest parties output the lock_j of all other honest parties, at least with grade 0. Moreover, B_{cand} is chosen to be the block with the greatest height from the output of $\mathsf{GA}_{0,j}$. Since, the outputs of $\mathsf{GA}_{0,j}$ do not conflict, B_{cand} set by all honest parties awake at time $(10e_j + 6)\Delta_j$ must either be equal to lock_j or a block that extends lock_j for all honest parties that are awake at that time. This proves the first part of the lemma.

To prove the second part of the lemma, recall that parties choose as B_{cand} their highest output of any grade from $\mathsf{GA}_{1,j}$. The integrity property implies that some honest party must have input a block extending B_{cand} to $\mathsf{GA}_{1,j}$ at time $(10e_j + 3)\Delta_j$. Honest parties choose as B_{cand} their highest grade 1 output from $\mathsf{GA}_{0,j}$ (with no conflicting outputs). The graded delivery property ensures that if an honest party outputs a block with grade 1, all honest parties output it with at least grade 0.

□

Lemma 123. *If Δ_j holds, no honest party has est $> j$, $j < k$, and all honest parties awake at time t have decided B (or a block extending it) in π_j by time t, then all honest parties awake at any point at or after t have lock_{j+1} equal to a block extending B for all epochs of π_{j+1} beginning at or after time t.*

Proof. By conditional safety (Lemma 121), no honest party decides a block conflicting with B in π_j. By conditional safety and Lemma 119, it follows that no honest party ever has lock_{j+1} conflicting with B. From the way that parties choose their inputs to $\mathsf{GA}_{0,j+1}$ on lines 9- 10, it follows that all honest inputs to $\mathsf{GA}_{0,j+1}$ of π_{j+1} are B or a block extending it for all epochs beginning at or after time t.

In every epoch of π_{j+1} starting after time t, by the validity and integrity properties of GA, no honest party outputs a block conflicting with B from $\mathsf{GA}_{0,j+1}$ and all honest parties that output from $\mathsf{GA}_{0,j+1}$ (i.e. all honest parties that are awake or in the recovering state at that time) do so with B (or a block extending it) with grade 1. As a result, all awake honest parties input a block extending B to $\mathsf{GA}_{1,j+1}$.The lemma therefore follows for lock updates prior to GPE due to the integrity and validity properties of GA. By the condition on line 37 of the joining protocol, honest parties don't become awake in the middle of any epoch. It follows that all honest parties awake at any point during the GPE of that epoch will have lock_{j+1} equal to a block extending B. By the way honest parties evaluate $\mathsf{predicate}()$, the lemma follows for lock updates after GPE due to integrity. □

Lemma 124. *If there are 2κ consecutive epochs in π_j where all honest parties awake at any point during that time period have est $= j$ and Δ_j holds, then during at least κ of those epochs all honest parties awake or recovering by the end of that epoch decide B, where B is the input of an honest*

party, with probability $1 - e^{-2\kappa\epsilon^2}$[8]*. In addition, during the time in which all awake honest parties have* $est = j$*,* $\forall j' > j$ *there is at least one epoch out of every* κ *in which all of the honest parties awake by the end of that epoch output* B*, where* B *is the input of an honest party, with probability* $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$*.*

Proof. We start by proving the first statement. By Lemma 122, and line 19 of the full protocol, in each such epoch of π_j, the inputs of all honest parties to GPE satisfy *predicate* for all honest parties that are awake at any point during the GPE (this holds because honest parties don't become awake in the middle of any epoch, by the condition on line 37 of the joining protocol). As such, in expectation, a block that is the input of an honest party to GPE is decided every $\frac{1}{\frac{1}{2}+\epsilon}$ epochs in expectation by the validity property of GPE by all honest parties that are awake or recovering by the end of that epoch. With the application of a Chernoff Bound, it follows that with probability $1 - e^{-2\kappa\epsilon^2}$, this occurs at least κ epochs out of every 2κ.

We now address the second statement in the lemma. Again, by Lemma 122, the inputs of all awake honest parties to GPE satisfy *predicate* for all honest parties awake during GPE in every epoch where all honest parties have $est = j$. The remainder of the lemma therefore follows from a simple Chernoff Bound. □

Lemma 125. *If an honest party* p_i *decides block* B *directly in epoch* e_j *of* π_j*,* Δ_j *holds, and all honest parties have* $est < j$ *then all honest parties that are awake at the end of epoch* $e_j + \kappa$ *will have decided* B *in* π_j *with probability* $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$*.*

Proof. Because p_i decided B in π_j at time t, it follows from Lemma 120 that all honest parties awake at time t' after the end of epoch e_j have lock_j equal to a block extending B from time t' onward. By conditional safety (Lemma 121), no honest party decides a block conflicting with B in π_j. Furthermore, by Lemma 122 and the fact that all honest parties have $est < j$, in all subsequent epochs the blocks input to GPE by honest parties satisfy *predicate* for all honest parties awake during that GPE (and therefore extend B).

[8] Note that a recovering party may decide B while it is simulating the protocol.

From the validity property of GPE, it follows that with probability $\frac{1}{2} + \epsilon$ all awake honest parties output $(B', 1)$ from GPE (and therefore decide it) where B' is the input of an honest party and therefore extending B in each such epoch. It follows with the application of a Chernoff Bound that this occurs within κ epochs with probability $1 - 2^{-\kappa} e^{-2\kappa\epsilon}$. □

Lemma 126. *If an honest party sets est* $:= j$ *at time* t *and no honest party ever has* $est + move > j$*, then all honest parties awake at time* $t' \geqslant t + 3\Delta_k$ *onward have* $est = j$ *by time* t'*.*

Proof. On line 6 of the joining protocol, a joining party sets its values of *est* and *move* for the first time during the joining protocol based on one iteration of π_{beacon}. Since no honest party ever has $est + move > j$, no honest party can ever output $j' > j$ from the GA in π_{beacon} by integrity, so no honest party can ever have $est > j$. It follows that if all honest parties awake at any point $t'' > t$ input j to π_{beacon}, then all honest parties awake at any time $t' \geqslant t + 3\Delta_k$ onward will have $est = j$ by time t'. We show that this holds.

By graded delivery, all honest parties awake or in the recovering state output j with grade 0 or 1 and no honest parties output $j' > j$ by integrity from GA in π_{beacon} that begins at time $t - 3\Delta_k$. From lines 4- 8 of π_{beacon}, it follows that all awake honest parties input to the GA of the subsequent iteration of π_{beacon} j. It follows from validity that all honest parties that output from the iteration of π_{beacon} that ends at time $t + 3\Delta_k$ output $(j, 1)$, and all honest parties that are awake have $est = j$ and $move = 0$ at this point by lines 4- 6 of π_{beacon}. Since joining parties set their starting values of *est* and *move* based on the output of an iteration of π_{beacon} prior to becoming awake, it follows that any party that is awake and therefore inputs to the next iteration of π_{beacon} has $est = j$ at $move = 0$ at the time of doing so. This continues to hold for all subsequent iterations of π_{beacon} due to the fact that joining parties set their starting values of *est* and *move* based on the output of an iteration of π_{beacon} prior to becoming awake and the fact that honest parties never set *move* to a value less than 0. □

Lemma 127. *No honest party ever sets* $est > ideal$ *except for probability* $1 - \mathsf{negl}(\kappa)$.

Proof. From the protocol in Figure 4.3, an honest party must output a value from GA_B that is greater than *ideal* in order to set *est* past *ideal*. By the integrity property of GA, this implies

that some honest party input a value $j > ideal$ to GA. To do so, that party must already have $est > ideal$ or $est \leqslant ideal$ and $move \geqslant 1$.

Now, we show a proof by contradiction. Let p be the first party to set $est > ideal$ (if multiple did at the same time, choose one arbitrarily without loss of generality), and let t be the time at which they did so. If p set $est > ideal$ while simulating the protocol based on saved messages during the joining protocol, let t be the time at which the GA in π_{beacon} that caused them to do so ended. This implies three possibilities prior to time t due to the integrity property of GA: (a) some awake honest party p' had $est = ideal$ and set $move := 1$ due to meeting one of the criteria for moving up or (b) some honest party had $est \leqslant ideal$ and $move > ideal - est$ due to outputting a value $v > ideal$ from GA_B in π_{beacon} or (c) some honest party joined the protocol with $est = ideal$ and $move = 1$.

We start by addressing the first case, and check if it is possible for some party p' to have had $est = ideal$ and set $move := 1$ due to meeting one of the criteria for moving up. Without loss of generality, let p' be the first party to have $est = ideal$ and $move := 1$. We go through each possible criteria for moving up to check if it's possible for p' to meet one of them at some time t^* prior to time t, assuming that none of the other criteria are met prior to time t^*. If none of the criteria are met in the absence of any of the other criteria already having been met, it follows that possibility (a) could not have happened.

We start with criterion 1 for moving up, and let $t' \leqslant t^* < t$ be the time at which p' set $est = ideal$. All honest parties awake at time $t' + 3\Delta_k$ onwards must have had $est = ideal$ by time $t' + 3\Delta_k$ by Lemma 126. By Lemma 124 there are κ epochs out of every 2κ epochs from time $t' + 3\Delta_k$ onward in which all honest parties that are awake or recovering by the end of that epoch decide a new block that is the input of an honest party with probability $1 - e^{-2\kappa\epsilon^2}$, and this holds for every 2κ epoch period after that (as long as no honest party increments est past $ideal$ for any other reason).

Next, we address criterion 2. Again, let $t' \leqslant t^* < t$ be the time at which p' set $est = ideal$. All honest parties awake at time $t'' \geqslant t' + 3\Delta_k$ onward must have had $est = ideal$ by time t'' by Lemma 126 (prior to which this criterion could not have been met for p'). By Lemma 125, when an honest party directly decides a block in π_j for $j \geqslant ideal$ and all awake honest

parties have $est = ideal$, all honest parties awake by the end of the subsequent κ epochs will have decided that block with probability $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$. By Lemma 123, once all honest parties awake at a certain time τ decide a block B_τ in $\pi_{j \geqslant ideal}$, all honest parties awake at any time $\tau' \geqslant \tau$ are locked on a block extending B_τ in π_{j+1} for all epochs of π_{j+1} beginning at or after time τ. By Lemma 122 and Lemma 124, all honest parties awake at the end of the κ subsequent epochs of π_{j+1} will have decided that block with probability $1 - 2^{-\kappa}e^{-2\kappa\epsilon}$. If a party decides a block indirectly during the joining protocol in protocol π_j, this implies that some honest party must have previously decided this block directly by the integrity property of GA. It follows from the fact that $\Delta_j = 2\Delta_{j-1}\ \forall 1 < j \leqslant k$ that any block decided in protocol π_j for $ideal \leqslant j < k$ all honest parties awake at the end of the subsequent 2κ epochs of $\pi_{j'+1}$ will have decided that block with high probability, in which case the criterion cannot be met in the absence of any of the other criterion having been met.

Finally, we address criterion 3. We start by checking whether the first part of the statement is possible: $\mathsf{lock}_{j'}$ conflicts with the chain constructed in π_j for some $j \geqslant ideal$ and $j' > j$. Conditional safety (Lemma 121) applies for π_j since no honest party has $est > ideal$, and Lemma 119 ensures that $\mathsf{lock}_{j'}$ was decided by an honest party in π_j. It follows that this part of the criterion cannot be met. The second part of the statement, that there is a fork in the chain constructed in π_j for $j \geqslant ideal$, also cannot be met due to conditional safety.

It therefore follows that option (a) could not have happened.

Next, we address (b): the possibility that p' had $est \leqslant ideal$ and $move > ideal - est$ due to outputting a value $v > ideal$ from GA in π_{beacon} prior to time t. For this to happen, p' must have output $v \geqslant ideal + 1$ from an instance of GA in π_{beacon} prior to time t. By the integrity property of GA, some honest party p'' must have input $v' \geqslant v$ to GA prior to time t. Without loss of generality, assume that p'' is the first party to have done so prior to time t. As already addressed when exploring possibility (a), it is not possible for p'' to have done so due to having $est = ideal$ and $move = 1$ from meeting the criteria for moving up except for negligible probability. Assume for now that (c), p'' joining the protocol with $est = ideal$ and $move = 1$, also cannot happen (we will later show that this is the case). By the definition of p'', possibility (b) also cannot be because it would imply that a party input $v'' \geqslant v$ to GA

prior to p'' doing so.

Finally, we address (c): the possibility that p' joined the protocol with $est = ideal$ and $move = 1$.

Assume first that p' recovered with $est = ideal$. By Lemma 118, it must be the case that $t^*_{1,ideal} \leqslant t^*_{ideal} \leqslant t^*_{1,ideal} + 3\Delta_k$. For p_{join} to set $move := 1$, we address three possibilities: (i) it must set $move := 1$ on line 31 due to having decided too few blocks to some chain Λ_j for $j \geqslant ideal$ on line 19 (ii) it meets criteria 1, 2, or 3 for moving up or (iii) there are conflicting outputs (of any grade) in $\mathsf{out}_{high,j}$ for $j \geqslant est$ (line 14).

We start by addressing (i). By time ts, $\lfloor \frac{ts-(t^*_{1,ideal}+3\Delta_k)}{20\kappa\Delta_{ideal}} \rfloor$ 2κ epoch periods must have elapsed in which there were κ epochs such that a new block was decided by all honest parties awake at the end of that epoch in π_{ideal}. By the integrity property of GA, the way that honest parties set $\mathsf{backlog}_{ideal}$ on line 4 of π_{beacon}, and Lemmas 121 and 124, it follows that at least $\mathsf{backlog}_{ideal} + \max(\lfloor \frac{ts-(t^*_{1,ideal}+3\Delta_k)}{20\kappa\Delta_{ideal}} \rfloor \cdot \kappa, 0)$ blocks must have been decided in π_{ideal} by all honest parties awake at time ts. By Lemma 114, it follows that p_{join} decides all blocks that were decided by all honest parties awake at time ts in π_{ideal}. For all $j > ideal$, the only way for p_{join} to set $move := 1$ on line 31 is if more than $3\Delta_k + \sum_{i=est}^{j} 20\kappa\Delta_i$ time has passed since t^*_{est} and fewer than $\mathsf{backlog}_{est} + min\_blocks_j$ blocks were decided. By Lemma 106, $\mathsf{backlog}_{est} + min\_blocks_j$ blocks must have been decided by time ts by all awake honest parties that were awake at time ts in π_j and their logs must not conflict by Lemma 121. By Lemma 114, p_{join} must have decided those blocks in π_j as well on line 19. It follows that p_{join} does not set $move := 1$ for this reason.

Next, we address (ii), and we start with criterion 1. For criterion 1 not to be triggered, p_{join} must see κ blocks decided every 2κ epochs in π_{ideal} from time $t_c = t^*_{est} + 3\Delta_k + 20\kappa\Delta_{est}\lfloor \frac{ts-(t^*_{est}+3\Delta_k)}{20\kappa\Delta_{ideal}} \rfloor$ onward. If p_{join} decided an excess of $\mathsf{backlog}_{ideal} + min\_blocks_{ideal}$ blocks on line 19, then these count toward the blocks it sees in the first 2κ epoch period after t_c (as per line 39 and the way that it sets $ts_{B,ideal}$ for each of these blocks on line 33). If it didn't see an excess of min_blocks_{ideal} blocks decided, by Lemma 114 this implies that none of the κ epochs in the 2κ period starting at time t_c in which a new block would have been decided

by all honest parties has happened prior to time ts. By Lemma 124, except for negligible probability those κ epochs will occur in each 2κ epoch period after t_c, and p_{join} *will* decide those blocks since it starts receiving messages from the time it enters the *recovering* state, which is prior to time ts (and it will store the timestamps of those blocks based on the simulation). It follows from Lemma 124 that p_{join} will not meet criterion 1 for moving up for any of the 2κ epoch periods after t_c prior to setting its status to awake except for negligible probability.

For criterion 2, p_{join} starts checking this criteria for the blocks in $\text{diff}(\Lambda_j, \Lambda_{j+1})$ for $est \leqslant j < k$. By Lemma 114, it follows that because p_{join} has not decided such a block B in π_{j+1} yet, this is because the point at which it has been decided by *all* awake honest parties hasn't happened yet. For such a block, $ts_{B,j}$ (as set on line 36) must be lower bounded by the time at which the first honest party to decide B in π_j does so. This is because an honest party must have decided B and input a block extending B to $\mathsf{GA}_{high,j}$ for it to have been output by p_{join} by the integrity property of GA. It therefore follows from Lemma 124 that for any block B in $\text{diff}(\Lambda_j, \Lambda_{j+1})$ such that $ts_{B,j} > t^*_j$ for $j \geqslant est$, p_{join} *will* decide B in π_{j+1} within the subsequent 2κ epochs after $ts_{B,j}$, because the point at which it is decided by all awake parties is after ts, at which point p_{join} starts receiving all messages. Lemmas 120 and 121, and the integrity property of GA, ensure that criterion 3 will not be triggered.

Similarly, Lemmas 114 and 121, and the integrity property of GA ensure that (iii) will not happen. p_{join} cannot $move := 1$ with $est = ideal$ without recovering with $est = ideal$ prior to joining due to the same argument as case (a).

We have thus arrived at a contradiction. □

Definition 128 ($t^*_{1,j}$)**.** *$t^*_{1,j}$ is defined as the earliest time at which an honest party outputs $(j, 1)$ from π_{beacon}.*[9]

Definition 129 ($checkpoint_{i,t}$)**.** *$checkpoint_{i,t}$ for party p_i at time t is defined as decide_j for p_i s.t. $j \geqslant ideal$ and $\forall j' \geqslant ideal$, $height(\mathsf{decide}_j) \geqslant height(\mathsf{decide}_{j'})$.*

[9] Note that the honest party may do this while simulating the protocol based on received messages in the joining protocol, in which case $t^*_{1,j}$ is they time they would have output $(j, 1)$ had they been awake at the time of receiving those messages, rather than the timestamp during simulation.

Definition 130 (*min_backlog_height*$_j$)**.** *Let H_t be the set of honest parties awake at time $t = t^*_{1,j} - 3\Delta_k$. min_backlog_height$_j$ is defined as, for the set A of honest parties awake at any time from $t^*_{1,j} - 3\Delta_k$ onward that do not recover with $est = j$, checkpoint$_{i,t}$ s.t. checkpoint$_{i,t} \leqslant$ checkpoint$_{j,t'} \forall p_j \in A$ and $t' \geqslant t$.*

Recall from Section 4.4 that we say that a party *recovers with* $est = j$ if it sets $est = j$ if it sets $est = j$ during its first full iteration of π_{beacon} in the recovery protocol (i.e. the one beginning at time ts).

Lemma 131. *For any honest party that recovers with $est = j$, backlog$_j \geqslant$ min_backlog_height$_j$.*

Proof. This follows from the integrity property and validity properties of GA, and the fact that if an honest party recovers with $est = j$, they must have set $\mathsf{backlog}_j$ based on the output of $\mathsf{GA}_{backlog,j}$ beginning at time ts s.t. $ts \geqslant t^*_{1,j} - 3\Delta_k$ (otherwise they couldn't have recovered with $est = j$), and the way that honest parties choose their inputs to $\mathsf{GA}_{backlog,j}$. □

Lemma 132. *Blocks are eventually decided at a rate of 1 block per $O(\Delta_{ideal})$ time in expectation.*

Proof. To prove the lemma statement, we show that in all π_j for $ideal \leqslant j \leqslant k$, $2^{j-ideal}\kappa$ blocks are decided every 2κ epochs. Because $\Delta_j = \frac{\Delta_{j+1}}{2}$ $\forall j < k$, this implies the lemma statement.

First, we prove the claim that new blocks are decided at a rate of κ blocks every 2κ epochs in π_{ideal}. By Lemma 127, we know that all honest parties have $est \leqslant ideal$ for the duration of the protocol. If there eventually comes a time that all awake honest parties have $est = ideal$, the claim follows from Lemma 124. If this does not eventually happen, validity of GA implies that for every iteration of π_{beacon}, some honest party inputs $est + move < ideal$. Eventually, by validity of GPE, there must be some $j < ideal$ such that some honest party (note that it need not be the same one each time) inputs j to each iteration of π_{beacon} for the remainder of the protocol. Otherwise, at some point, all the awake honest parties *would* reach $est = ideal$.

As per Definition 128, let $t^*_{1,j}$ be the earliest time at which an iteration of π_{beacon} output $(j, 1)$ for an honest party. By graded delivery and validity of GA and the fact that honest

parties listen to (more than) one iteration of π_{beacon} prior to becoming awake, all honest parties awake at time $t^*_{2,j} = t^*_{1,j} + 3\Delta_k$ must have output $(j, 1)$ by that time. Note that no honest party could output $(j', 1)$ for $j' > j$ from π_{beacon}, as this would contradict the definition of j.

Let t be some time during the course of the protocol s.t. $t > t^*_{2,j}$. Let p be the party that input j to the latest instance of π_{beacon} that completed at or prior to time t with $move = 0$ (note that some honest party must have done this, otherwise validity of GA would imply that all of the awake honest parties output $(j + 1, 1)$ from the subsequent GA). If p did not recover with $est = j$ (i.e. it was awake at the time of doing so or did so while simulating the protocol), this implies that it didn't meet any of the criteria for moving up. That is, it decided κ blocks for every 2κ epochs from $t^*_j \leqslant t^*_{2,j}$ onward and every block it decided in $\pi_{j'}$, it decided in $\pi_{j'+1}$ for $j < j' < k$ within the 2κ subsequent epochs. In this case, the lemma follows from Lemma 125 (which shows that all honest parties decide the blocks decided by p in $\pi_{j'' \geqslant ideal}$ with a small delay but at the same latency) and the fact that $\Delta_{j+1} = 2\Delta_j \forall j < k$.

Otherwise, if p recovered with $est = j$, this implies that it did not set $move := 1$ on line 31 of the joining protocol. That is, the length of its log Λ_j is at least $min\_blocks_j + min\_backlog\_height_j$, which is equivalent to having decided $min\_backlog\_height_j$ blocks prior to t^*_j and having decided blocks at a rate of κ blocks for every 2κ epoch of π_j from time t^*_j onward. Note that by Lemma 118, $t^*_j \leqslant t^*_{1,j} + 3\Delta_k$. Furthermore, for all $\pi_{j'>j}$ the length of $\Lambda_{j'}$ is at least $\mathsf{backlog}_j + min\_blocks_{j'-1} - 2^{j'-j}\kappa$, implying that $\Lambda_{j'}$ is growing at the rate of κ blocks per every 2κ epochs of π_j from time t^*_j with a small, constant, bounded delay. Furthermore, p has never met any of the criteria for moving up by the time that that it inputs to π_{beacon}, implying that it continued to decide the equivalent of κ blocks per 2κ epochs if π_j from time t^*_j onward up to time t, and that those blocks were confirmed in higher sub-protocols. Because every honest party that joins the protocol has $t^*_j \leqslant t^*_{1,j} + 3\Delta_k$, if an honest party recovers with $est = j$ and $move := 0$, the above argument, Lemmas 125 and 131, and the fact that $\Delta_{j+1} = 2\Delta_j \forall j < k$ imply that the lemma holds.

□

Lemma 133 (Liveness)**.** *Blocks containing transactions continue to be appended to* $\Lambda_j \forall j \geqslant ideal$*.*

Proof. The lemma follows from Lemma 132. □

Lemma 134 (Safety)**.** *If two honest parties deliver logs* $\Lambda_{j'}$ *and* $\Lambda'_{j'}$*, then* $\Lambda_{j'}$ *and* $\Lambda'_{j'}$ *do not conflict* $\forall \Delta_{j'} \geqslant \Delta_{ideal}$*.*

Proof. The lemma follows from conditional safety (Lemma 121) and the fact that no honest party ever has $est > ideal$ (Lemma 127). □

$B_0 :=$ Genesis block
$\mathsf{lock}_j := \mathsf{decide}_j := \mathsf{backlog}_j := B_0$
$est := 1$
$move := 0$
$t_1^* := 0$
$t_{j \in 2,\ldots,k}^* := \infty$
for epochs $e_j = 0, 1, 2, 3, \ldots$ **do**
  at $10e_j\Delta_j$ **do** ▷ Run $\mathsf{GA}_{0,j}$
    if $est < j$ and $j > 1$ and decide_{j-1} extends lock_j **then** $\mathsf{inp}_{0,j} := \mathsf{decide}_{j-1}$
    else $\mathsf{inp}_{0,j} := \mathsf{lock}_j$
    $\mathsf{out}_{0,j} := \mathsf{GA}_{0,j}(\mathsf{inp}_{0,j})$ ▷ $\mathsf{out}_{0,j}$ is a set of tuples (B^*, g) denoting a block and its grade
    $B_0 :=$ the block with the greatest height output with grade 1 such that it does not conflict with any block output from $\mathsf{out}_{0,j}$
  at $(10e_j + 3)\Delta_j$ **do** ▷ Run $\mathsf{GA}_{1,j}$
    $\mathsf{out}_{1,j} := \mathsf{GA}_{1,j}(B_0)$
    $\mathsf{lock}_j :=$ block with the greatest height output with grade 1 from $\mathsf{out}_{1,j}$
    $B_{\mathsf{cand}} :=$ block with the greatest height output from $\mathsf{out}_{1,j}$
  at $(10e_j + 6)\Delta_j$ **do** ▷ Run GPE
    if $est = j$ **then**
      $B_{\mathsf{proposal}} := (txns, H(B_{\mathsf{cand}}))$ ▷ Create a new block proposal
      $(B_{\mathsf{proposed}}, g) := \mathsf{GPE}_j(B_{\mathsf{proposal}}, \mathsf{predicate}(B) = \mathsf{true}$ if B extends $\mathsf{lock}_j)$
    else
      $B_{\mathsf{proposal}} := B_{\mathsf{cand}}$
      $(B_{\mathsf{proposed}}, g) := \mathsf{GPE}_j(B_{\mathsf{proposal}}, \mathsf{predicate}(B) = \mathsf{true}$ if B extends lock_j and a block extending B was output from $\mathsf{GA}_{0,j})$
    if $B_{\mathsf{proposed}} \neq \perp$ **then**
      $\mathsf{lock}_j := B_{\mathsf{proposed}}$
      if $g = 1$ **then** $\mathsf{decide}_j := B_{\mathsf{proposed}}$ (if haven't decided it already)
  at any time **do**
    upon deciding block B in π_j **do** set $e_{B,j} := e_j$ and $ts_{B,j} := ts_{now}$
    if any of the criteria for moving up is met **then**
      cease participation in π_{est} and set $move := 1$

Criteria for moving up:

1. **No progress in π_{est}.** Beginning at time $t_{est}^* + 3\Delta_k$ or later, a 2κ epoch period elapses in π_{est} during which fewer than κ blocks are decided
2. **No confirmations in higher sub-protocols.** At time $t_{est}^* + 3\Delta_k$ or later for any j' s.t. $est \leqslant j' < k$ a block B is decided in $\pi_{j'}$ and more than 2κ full epochs of $\pi_{j'+1}$ elapse after $ts_{B,j'}$ without B having been decided in $\pi_{j'+1}$
3. **Observing equivocations in higher sub-protocols.** $\mathsf{lock}_{j'}$ conflicts with the chain constructed in π_j for some $j' > j$ or there is a fork in the chain constructed in π_j for some $j \geqslant est$

FIGURE 4.2: Protocol π_j for party P_i.

1: **at** time $t = 3\Delta_k, 6\Delta_k, 9\Delta_k, \ldots$ **do**
2: out $:=$ GA$(est + move)$
3: let $v = val$ and $g = grade$ s.t. $(val, grade) \in$ out and $\forall (val', grade') \in$ out, $val \geqslant val'$ (if val was output more than once, choose the highest $grade$)
4: **if** $v > est$ and $g = 1$ **then**
5: cease participation in all sub-protocols π_j for $j < v$
6: set $est := v$, $move := 0$, $t^*_{est} := ts_{now}$, and
7: **else if** $v > est$ and $g = 0$ **then**
8: set $move := v - est$

FIGURE 4.3: Protocol π_{beacon} for party P_i run with delay Δ_k.

//*awake* parties:

1: **at** any time **do**
2: **upon** updating est to j for the first time **do**
3: set $\mathsf{backlog}_{est}$:= $height(\mathsf{decide}_j)$ s.t.:
4: $j \geqslant est$ and $\forall j' > j$: decide_j does not conflict with $\Lambda_{j'}$ and $height(\mathsf{decide}_j) \geqslant height(\mathsf{decide}_{j'})$
5: **at** time $t = 3\Delta_k, 6\Delta_k, 9\Delta_k, \ldots$ **do**
6: **for** $j = 1, \ldots, k$ **do**
7: $\mathsf{out}_{high,j} := \mathsf{GA}_{high,j}(\mathsf{decide}_j)$, $\mathsf{out}_{lock,j} := \mathsf{GA}_{lock,j}(\mathsf{lock}_j)$ ▷ p_{join} stores output
8: $\mathsf{out}_{backlog,j}$:=$\mathsf{GA}_{backlog,j}$ ($\mathsf{backlog}_j$ **if** $j \leqslant est$ **else** $height(\mathsf{decide}_{j'})$) s.t.:
9: $j' \geqslant j$ and $\forall j'' > j'$: $\mathsf{decide}_{j'}$ does not conflict with $\Lambda_{j''}$ and $height(\mathsf{decide}_{j'}) \geqslant height(\mathsf{decide}_{j''})$

//p_{join}:

10: set status to *recovering* and begin storing all messages received and the time they were received
11: send $\langle \mathsf{recovering}, p_{join}, ts \rangle_{join}$ to all parties, where ts is the timestamp of the start of the next full iteration of π_{beacon}
12: **at** time $t = ts + 3\Delta_k$ **do**
13: set est and $move$ according to the outputs of GA in the iteration of π_{beacon} starting at time ts according to lines 6 and 8
14: **if** $\exists j \geqslant est$ s.t. there are any two conflicting outputs (of any grade) in $\mathsf{out}_{high,j}$ **then**
15: set $move := 1$
16: set $B_{high,j\in[est,\ldots,k]}$:= the highest block in $\mathsf{out}_{high,j}$ (with any grade)
17: set $\mathsf{lock}_{j\in[est,\ldots,k]}$:= the highest block in $\mathsf{out}_{lock,j}$ (with any grade)
18: **for** $j \geqslant est$ **do**
19: set $\mathsf{decide}_j := B_{high,j}$

//*awake* parties:

20: **upon** receiving message $\langle \mathsf{recovering}, p_{join}, ts \rangle_{join}$ from party p_{join} **do**
21: **at** time $t = ts + 3\Delta_k$ **do**
22: send to p_{join} $\langle \mathsf{status}, est, t^*_{est} \rangle_i$

//p_{join}:

23: **at** time $t = ts + 4\Delta_k$ **do**
24: let P be the list of parties from whom status messages were received
25: let M_{est} be the list of values of t^*_{est} received from every party $p_i \in P$ (for parties that sent a lower est than p_{join}'s, use t_{now}, the current timestamp)
26: p_{join} sets $t^*_{est} := \text{median}(M_{est})$
27: $min\_blocks_{est} := \max(\lfloor \frac{ts-(t^*_{est}+3\Delta_k)}{20\kappa\Delta_{est}} \rfloor \kappa, 0)$
28: **for** $j = est + 1, \ldots, k$ **do**
29: set $min\_blocks_j := \max(min\_blocks_{j-1} - 2^{j-est}\kappa, 0)$
30: **if** $\exists j \geqslant est$ s.t. $ts > t^*_{est} + 3\Delta_k + \sum_{i=est}^{j} 20\kappa\Delta_i$ and $min\_blocks_j + \mathsf{backlog}_{est} > |\Lambda_j|$ **then**
31: set $move := 1$
32: **for** block $B \in \Lambda_{est}[\mathsf{backlog}_{est} + min\_blocks_{est} + 1 :]$ **do**
33: set $ts_{B,j} := ts$
34: **for** $j = est + 1, \ldots, k$ **do**
35: **for** block $B \in \Lambda_j$ **do**
36: set $ts_{B,j} := ts$
37: **upon** completion of an entire epoch of π_k since setting status to *recovering* **do**:
38: update state based on simulation of sub-protocols $\pi_{est,\ldots,k}$ and π_{beacon} using stored messages (except for updates to $move$ based on meeting criteria for moving up)
39: begin checking criterion 1 for moving up from time $t^*_{est} + 3\Delta_k + 20\kappa\Delta_{est}\lfloor \frac{ts-(t^*_{est}+3\Delta_k)}{20\kappa\Delta_{est}} \rfloor$ onward
40: begin checking criterion 2 for moving up for blocks in $\text{diff}(\Lambda_j, \Lambda_{j+1}) \forall est \leqslant j < k$
41: begin checking criterion 3 for moving up
42: set status to *awake*

FIGURE 4.4: Joining Protocol for *awake* parties p_i and *recovering* party p_{join} executed with delay of Δ_k

5. Conclusions

In this work, we have explored ways to design consensus protocols with better latency. For the problem of asynchronous agreement, we showed a simple framework for designing protocols that are easy to prove and efficient in the number of rounds until all parties terminate the protocol. In addition, we showed new upper and lower bounds for key primitives that are used to achieve asynchronous agreement. For the problem of leader election, we showed a protocol that is scalable, in that each party sends and processes only a polylogarithmic number of bits, ensuring that there is no bottleneck. Finally, in a network with dynamic participation, we showed a protocol that achieves latency better than the pessimistic network delay assumption, and do so with the introduction of a new model, generalized synchrony.

Appendix A. Crash Fault Tolerant Binding Crusader Agreement for Adaptive Inputs

In this section, we discuss our interest in crash fault tolerant protocols for binding crusader agreement that are secure even with *adaptive inputs*. By this, we mean that the binding property holds even if the adversary may adaptively choose the inputs of parties at any point in the execution of the protocol prior to scheduling their first actions. There are two advantages of using binding crusader agreement for adaptive inputs as a building block for crash fault tolerant asynchronous agreement protocols: efficiency and simplicity.

In [1], Abraham, Ben-David and Yandamuri show a simple framework for asynchronous agreement that uses a strong common coin (such that all parties see value $v \in \{0,1\}$ with probability $\frac{1}{2}$) and binding crusader agreement. Although the authors don't explicitly state it, the crash fault tolerant BCA protocol from [1] withstands an adversary that can adaptively choose the inputs of parties when they start the protocol. In fact, when this requirement is removed, we obtain the simpler 1 round protocol of Algorithm 1 for crash fault tolerant BCA. When the inputs of all parties are fixed prior to the start of the protocol, binding trivially follows from the fact that there is at most one value $v \in \{0,1\}$ such that $n - t$ parties start the protocol with value v. In fact, binding is only guaranteed in this protocol *if the inputs of all parties are fixed prior to the start of the protocol.*

To see why this matters, first we review how the asynchronous agreement protocol terminates with the original BCA protocol for adaptive inputs. With the original BCA protocol for adaptive inputs, the asynchronous agreement protocol takes at most 7 rounds of broadcast in expectation for all parties to terminate the protocol. This follows from a simple invariant: in any given round of the AA protocol, with probability $\frac{1}{2}$, the value of the common coin is equal to the value to which the adversary is bound in that round's BCA. In that case, all parties adopt the same value est, and they all decide that value in the next round in which the coin is again equal to that value. In other words, the original protocol requires a single good event to occur, which happens with constant probability in each round.

Now, consider what happens when we plug the BCA protocol from Algorithm 1 into the asynchronous agreement protocol of [1]. Since the BCA protocol is not binding when

the adversary can adaptively choose the inputs of parties, we can no longer apply the same invariant to ensure termination. This is because the adversary can lag a party behind in the previous round of the AA protocol and choose its input to the next round's BCA. In this case, to argue termination it is necessary that two independent good events occur in two consecutive rounds, resulting in an AA protocol that requires more rounds of broadcast till termination and a more complex proof than the one presented in [1].

Algorithm 1 Asynchronous Binding Crusader Agreement for Crash Faults with Static Inputs

Input: x
1: send $\langle \text{val}, x \rangle$ to all
2: **upon** receiving $\langle \text{val}, * \rangle$ messages from $n - f$ parties:
3: **if** all the messages contain the same value x, decide x
4: **else**, decide $\perp$

Appendix B. Binding Crusader Agreement with Threshold Signatures

We use the following interface for the threshold signature scheme:

- threshold-sign$_i(m)$: produces signature share by p_i on message m.
- share-validate(m, s_j, pk_j): validates that signature share s_j was produced by p_j on m.
- threshold-combine(m, S): combines a set S of signature shares from distinct parties for message m into an $O(1)$-sized signature where $|S| \geqslant k$ and $\forall s_j \in S$, share-validate$(m, s_j, pk_j) = true$.
- threshold-verify(m, σ): returns true if σ was a result of computing threshold-combine(m, S) where $|S| \geqslant k$ and share-validate$(m, s_j, pk_j) = true, \forall s_j \in S$.

Algorithm 2 Asynchronous Binding Crusader Agreement with Threshold Signatures instance id ($BCA_{Byz,TSig}$)

Input: x

$est = x$

$sentEcho2 = false$, $sentEcho3 = false$, $pendingEcho = \{\}$, $pendingEcho2 = \{\}$, $pendingEcho3 = \{\}$, $vals = \{\}$

send $\langle$echo, est, threshold-sign$_i$(echo, id, est)$\rangle$ to all

upon receiving a message $\langle$echo, v, $ts\rangle$ from p_j for the first time such that share-validate((echo, id, v), ts, p_j) = true

  add (v, ts) to $pendingEcho$

  if there exists a single value v' such that the set M of $\{(v', ts')|(v', ts') \in pendingEcho\}$ is of size at least $t + 1$ and $sentEcho2 = false$ **then**

    let S be the set $\{s|(*, s) \in pendingEcho\}$

    $\sigma_{\text{echo},id,v'} \leftarrow$threshold-combine($\langle$echo, $id, v'\rangle$, S)

    send $\langle$echo2, v', $\sigma_{\text{echo},id,v'}\rangle$ to all, $sentEcho2 = true$

upon receiving a message $\langle$echo2, v, $\sigma\rangle$ from p_j for the first time such that threshold-verify((echo, id, v), σ) = $true$

  if $sentEcho2 = false$ **then**

    send $\langle$echo2, v, $\sigma\rangle$ to all, $sentEcho2 = true$

  add (v, σ) to $pendingEcho2$

  if $|pendingEcho2| \geqslant n - t$ and $sentEcho3 = false$

    let M be the set $\{v|(v, *) \in pendingEcho2\}$ and let S be the set $\{s|(*, s) \in pendingEcho2\}$

    if $|M| > 1$ **then**

      send $\langle$echo3,$\perp$, S, $\perp\rangle$ to all

    else send $\langle$echo3, v, S, threshold-sign$_i$(echo3, id, v)$\rangle$ to all, where v is the single value in M

    $sentEcho3$ = $true$

upon receiving a message m where m=$\langle$echo3, v, S, $ts\rangle$ from p_j for the first time such that either $v = \perp$ or share-validate((echo3, id, v), ts, p_j)=$true$

  if $v = \perp$ **then** $vals = \{0, 1\}$

  else $vals = \{v\}$

  if $\forall v' \in vals$ there exists $\sigma' \in S$ s.t. threshold-verify((echo, id, v'), σ') = $true$ **then**

    add (v, ts) to $pendingEcho3$

    if $|pendingEcho3| \geqslant n - t$ **then**

      let M be the set of values $\{v|(v, *) \in pendingEcho3\}$

      if $|M| > 1$ **then** decide $\perp$

      else

        let S be the set $\{s|(*, s) \in pendingEcho3\}$, and let v be the single value in M

        $\sigma_{\text{echo3},id,v} \leftarrow$threshold-combine((echo3, id, v), S)

        decide v

We now prove Theorem 49.

Proof of Theorem 49. The theorem follows from lemmas 135-139. □

Lemma 135. *Algorithm 2 satisfies agreement.*

Proof. Assume that p_i decides non-$\perp$ value v_i. Then p_i received $\langle$echo3, v_i, S, $ts\rangle$ from at least $n - t$ parties p_k such that share-validate((echo3, id, v_i), ts, p_k) = $true$. Since a non-faulty party sends a single echo3 message, by quorum intersection either $v_j = \perp$ or $v_i = v_j$. □

Lemma 136. *Algorithm 2 satisfies validity.*

Proof. Assume, for a contradiction, that all non-faulty parties input v to the protocol for instance id, and some non-faulty party p_i decides $v' \neq v$ (v' must be either $1 - v$ or $\perp$). p_i must have received a message $\langle$echo3, v', S', $ts\rangle$ from at least one party such that there is a $\sigma' \in S'$ and threshold-verify((echo, id, v'), σ') = $true$. For such a σ' to exist, that would require at least $t + 1$ parties p_j to have sent $\langle$echo, v', threshold-sign$_j$(echo, id, v')$\rangle$ messages, at least one of which would have to be non-faulty. Non-faulty parties only send a single echo message, and only for the value that they started with. Therefore, we have arrived at a contradiction. □

Lemma 137. *Algorithm 2 satisfies termination within 3 communication rounds.*

Proof. It is clear from examining the protocol that a party sends at most three messages: echo, echo2, and echo3.

Since we are considering the binary case, there must be some value v such that at least $t + 1$ non-faulty parties start with v. Therefore, eventually every non-faulty party receives at least $t + 1$ $\langle$echo, v, $ts\rangle$ from non-faulty parties p_i, where ts is p_i's output from running threshold-sign$_i$(echo, id, v). If they haven't already, every non-faulty party then sends an echo2 message containing a valid threshold signature σ on (echo, id, v). Every party then receives echo2 messages containing valid threshold signatures from at least $n - t$ parties and therefore has at least $n - t$ entries in its *pendingEcho*2 set. Each non-faulty party then sends an echo3 message accordingly, containing the set of threshold signatures needed for the value included in its message, along with a threshold signature share if it sends a non-$\perp$ value. Finally, each party receives at least $n - t$ such echo3 messages and decides a value. □

Lemma 138. *If a non-faulty party sends* $\langle$echo3, v, *, *$\rangle$*, where* $v \neq \perp$ *and another non-faulty party sends* $\langle$echo3, v', *, *$\rangle$*, either* $v' = v$ *or* $v' = \perp$.

Proof. Assume, for a contradiction, that non-faulty party p_i sends $\langle$echo3, v_i, *, *$\rangle$ and non-faulty party p_j sends $\langle$echo3, v_j, *, *$\rangle$, $v_i \neq v_j$, and both values are non-$\perp$. Then p_i must have at least $n - t$ $(v_i, *)$ in its *pendingEcho*2 set and p_j must have at least $n - t$ $(v_j, *)$ in its *pendingEcho*2 set. However, due to quorum intersection and the fact that a non-faulty party sends a single echo2 message, this is not possible. Thus, we have arrived at contradiction. □

Lemma 139. *Algorithm 2 satisfies binding.*

Proof. Let p_i be the first non-faulty party who decides. Consider the set of non-faulty parties of size at least $n - 2t \geqslant t + 1$ whose values sent in their echo3 messages are in p_i's *pendingEcho*3 set at the time at which it decides. If all of these non-faulty parties sent $\langle$echo3, $\perp$, *, *$\rangle$, then every non-faulty party has at least one such message in their *pendingEcho*3 set and therefore outputs $\perp$. If instead at least one of these non-faulty parties sent $\langle$echo3, b, *, *$\rangle$ where $b \neq \perp$, then, by Lemma 138, no non-faulty party sends $\langle$echo3, $1 - b$, *, *$\rangle$ and no non-faulty party can decide $1 - b$. □

Lemma 140. *The* $AA_{\frac{1}{2}}$*-*$BCA_{Byz,TSig}$ *protocol takes, in expectation, 13 rounds to terminate with a strong* $2t$*-resilient common coin.*

Proof. In an instance of $BCA_{Byz,TSig}$, parties perform exactly three broadcasts. Since the common coin has degree $2f$, the messages to reveal the coin can be sent with echo3 messages. It takes, in expectation, 2 rounds of $AA_{\frac{1}{2}}$-$BCA_{Byz,TSig}$ for all non-faulty parties to adopt the same value v and an additional 2 rounds in expectation until the coin value is again v. As a result, the protocol takes 12 broadcasts until all non-faulty parties commit, in expectation. Since every non-faulty party obtains a $\sigma_{echo3,id,v}$ when they decide v in instance id, they can forward this proof to all of the other parties and terminate because parties can decide v after receiving it. As a result, the protocol requires a single additional message for all parties to terminate, resulting in 12+1=13 broadcasts in expectation. □

Bibliography

[1] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. "Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement". In: *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*. 2022, pp. 381–391.

[2] Ittai Abraham et al. "Reaching Consensus for Asynchronous Distributed Key Generation". In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. PODC'21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 363–373. ISBN: 9781450385480. DOI: 10.1145/3465084.3467914. URL: https://doi.org/10.1145/3465084.3467914.

[3] Marcos K Aguilera et al. "Microsecond consensus for microsecond applications". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 2020, pp. 599–616.

[4] Marcos K Aguilera et al. "Ubft: Microsecond-scale bft using disaggregated memory". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 862–877.

[5] Marcos Kawazoe Aguilera and Sam Toueg. "A Simple Bivalency Proof that t-Resilient Consensus Requires $t + 1$ Rounds". In: *Inf. Process. Lett.* 71.3-4 (1999), pp. 155–158. DOI: 10.1016/S0020-0190(99)00100-3. URL: https://doi.org/10.1016/S0020-0190(99)00100-3.

[6] Marcos Kawazoe Aguilera and Sam Toueg. "Correctness Proof of Ben-Or's Randomized Consensus Algorithm". In: (1998).

[7] Vivek Bagaria et al. "Prism: Deconstructing the Blockchain to Approach Physical Limits". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 585–602. ISBN: 9781450367479. DOI: 10.1145/3319535.3363213. URL: https://doi.org/10.1145/3319535.3363213.

[8] Naama Ben-David and Kartik Nayak. "Brief announcement: Classifying trusted hardware via unidirectional communication". In: 2021, pp. 191–194.

[9] 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*. 1983, pp. 27–30.

[10] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. "Brief Announcement: Simple Gradecast Based Algorithms". In: *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*. Ed. by Nancy A. Lynch and Alexander A. Shvartsman. Vol. 6343. Lecture Notes in Computer Science.

Springer, 2010, pp. 194–197. DOI: 10.1007/978-3-642-15763-9_18. URL: https://doi.org/10.1007/978-3-642-15763-9%5C_18.

[11] Gabriel Bracha. "Asynchronous Byzantine agreement protocols". In: *Information and Computation* 75.2 (1987), pp. 130–143.

[12] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 335–350.

[13] Christian Cachin, Klaus Kursawe, and Victor Shoup. "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography". In: *in Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC*. 2000, pp. 123–132.

[14] Christian Cachin and Luca Zanolini. "From Symmetric to Asymmetric Asynchronous Byzantine Consensus". In: *arXiv preprint arXiv:2005.08795* (2020).

[15] Christian Cachin et al. "Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems". In: *in Proc. 9th ACM Conference on Computer and Communications Security (CCS*. ACM Press, 2002, pp. 88–97.

[16] Brad Calder et al. "Windows azure storage: a highly available cloud storage service with strong consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 143–157.

[17] 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, May 16-18, 1993, San Diego, CA, USA*. Ed. by S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal. ACM, 1993, pp. 42–51. DOI: 10.1145/167088.167105. URL: https://doi.org/10.1145/167088.167105.

[18] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. "A big data modeling methodology for Apache Cassandra". In: *2015 IEEE International Congress on Big Data*. IEEE. 2015, pp. 238–245.

[19] Shir Cohen, Idit Keidar, and Alexander Spiegelman. "Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp". In: 2020.

[20] James C Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22.

[21] 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". In: *arXiv preprint arXiv:2002.08765* (2020).

[22] 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)*. 2021, pp. 466–483. DOI: 10.1109/SP40001.2021.00087.

[23] Tyler Crain et al. "DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548057.

[24] Francesco D'Amato, Giuliano Losa, and Luca Zanolini. "Asynchrony-Resilient Sleepy Total-Order Broadcast Protocols". In: *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing, PODC 2024, Nantes, France, June 17-21, 2024*. Ed. by Ran Gelles, Dennis Olivetti, and Petr Kuznetsov. ACM, 2024, pp. 247–256. DOI: 10.1145/3662158.3662779. URL: https://doi.org/10.1145/3662158.3662779.

[25] Francesco D'Amato and Luca Zanolini. "Recent Latest Message Driven GHOST: Balancing Dynamic Availability With Asynchrony Resilience". In: *37th IEEE Computer Security Foundations Symposium, CSF 2024, Enschede, Netherlands, July 8-12, 2024*. IEEE, 2024, pp. 127–142. DOI: 10.1109/CSF61375.2024.00001. URL: https://doi.org/10.1109/CSF61375.2024.00001.

[26] Francesco D'Amato et al. "No More Attacks on Proof-of-Stake Ethereum?" In: *CoRR* abs/2209.03255 (2022). URL: https://doi.org/10.48550/arXiv.2209.03255.

[27] Francesco D'Amato et al. *TOB-SVD: Total-Order Broadcast with Single-Vote Decisions in the Sleepy Model*. 2024. arXiv: 2310.11331 [cs.DC]. URL: https://arxiv.org/abs/2310.11331.

[28] Sourav Das et al. *Practical Asynchronous Distributed Key Generation*. Cryptology ePrint Archive, Report 2021/1591. https://ia.cr/2021/1591. 2021.

[29] Richard A. DeMillo, Nancy A. Lynch, and Michael Merritt. "Cryptographic Protocols". In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*. Ed. by Harry R. Lewis et al. ACM, 1982, pp. 383–400. DOI: 10.1145/800070.802214. URL: https://doi.org/10.1145/800070.802214.

[30] Dan Dobre and Neeraj Suri. "One-step consensus with zero-degradation". In: *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE. 2006, pp. 137–146.

[31] Danny Dolev. "The Byzantine generals strike again". In: *Journal of algorithms* 3.1 (1982), pp. 14–30.

[32] Danny Dolev and Rüdiger Reischuk. "Bounds on information exchange for Byzantine agreement". In: *Journal of the ACM (JACM)* 32.1 (1985), pp. 191–204.

[33] Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. "Early Stopping in Byzantine Agreement". In: *J. ACM* 37.4 (1990), pp. 720–741. DOI: 10.1145/96559.96565. URL: https://doi.org/10.1145/96559.96565.

[34] Danny Dolev and H. Raymond Strong. "Authenticated Algorithms for Byzantine Agreement". In: *SIAM J. Comput.* 12.4 (1983), pp. 656–666. DOI: 10.1137/0212045. URL: https://doi.org/10.1137/0212045.

[35] Danny Dolev and H. Raymond Strong. "Authenticated Algorithms for Byzantine Agreement". In: *SIAM J. Comput.* 12.4 (1983), pp. 656–666. DOI: 10.1137/0212045. URL: https://doi.org/10.1137/0212045.

[36] 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*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2028–2041. ISBN: 9781450356930. DOI: 10.1145/3243734.3243812. URL: https://doi.org/10.1145/3243734.3243812.

[37] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (Apr. 1988), pp. 288–323. ISSN: 0004-5411. DOI: 10.1145/42282.42283. URL: https://doi.org/10.1145/42282.42283.

[38] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323. DOI: 10.1145/42282.42283. URL: https://doi.org/10.1145/42282.42283.

[39] Cynthia Dwork and Yoram Moses. "Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures". In: *Inf. Comput.* 88.2 (1990), pp. 156–186. DOI: 10.1016/0890-5401(90)90014-9. URL: https://doi.org/10.1016/0890-5401(90)90014-9.

[40] Pesech Feldman and Silvio Micali. "An optimal probabilistic protocol for synchronous Byzantine agreement". In: 26.4 (1997), pp. 873–933.

[41] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. "Impossibility of distributed consensus with one faulty process". In: 32.2 (1985), pp. 374–382.

[42] Michael J. Fischer and Nancy A. Lynch. "A Lower Bound for the Time to Assure Interactive Consistency". In: *Inf. Process. Lett.* 14.4 (1982), pp. 183–186. DOI: 10.1016/0020-0190(82)90033-3. URL: https://doi.org/10.1016/0020-0190(82)90033-3.

[43] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121. URL: http://doi.acm.org/10.1145/3149.214121.

[44] Matthias Fitzi, Chen-Da Liu-Zhang, and Julian Loss. "A New Way to Achieve Round-Efficient Byzantine Agreement". In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. PODC'21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 355–362. ISBN: 9781450385480. DOI: 10.1145/3465084.3467907. URL: https://doi.org/10.1145/3465084.3467907.

[45] Matthias Fitzi et al. “Parallel Chains: Improving Throughput and Latency of Blockchain Protocols via Parallel Composition”. In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 1119. URL: https://api.semanticscholar.org/CorpusID:54014893.

[46] Eli Gafni and Giuliano Losa. “Brief Announcement: Byzantine Consensus Under Dynamic Participation with a Well-Behaved Majority”. In: *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy*. Ed. by Rotem Oshman. Vol. 281. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 41:1–41:7. URL: https://doi.org/10.4230/LIPIcs.DISC.2023.41.

[47] Ankit Gangwal, Haripriya Ravali Gangavalli, and Apoorva Thirupathi. “A survey of Layer-two blockchain protocols”. In: *Journal of Network and Computer Applications* 209 (2023), p. 103539.

[48] Juan A Garay et al. “Adaptively secure broadcast, revisited”. In: *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 2011, pp. 179–186.

[49] Seth Gilbert and Nancy A. Lynch. “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[50] Vipul Goyal, Hanjun Li, and Justin Raizes. “Instant Block Confirmation in the Sleepy Model”. In: *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 65–83. ISBN: 978-3-662-64330-3. DOI: 10.1007/978-3-662-64331-0_4. URL: https://doi.org/10.1007/978-3-662-64331-0_4.

[51] Guy Golan Gueta et al. “SBFT: A scalable and decentralized trust infrastructure”. In: *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE. 2019, pp. 568–580.

[52] Martin Hirt and Vassilis Zikas. “Adaptively secure broadcast”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2010, pp. 466–485.

[53] Jonathan Katz and Chiu-Yuen Koo. “On Expected Constant-Round Protocols for Byzantine Agreement”. In: *J. Comput. Syst. Sci.* 75.2 (Feb. 2009), pp. 91–112. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2008.08.001. URL: https://doi.org/10.1016/j.jcss.2008.08.001.

[54] Leslie Lamport. “Fast paxos”. In: *Distributed Computing* 19 (2006), pp. 79–103.

[55] Leslie Lamport. “Lower bounds for asynchronous consensus”. In: *Distributed Comput.* 19.2 (2006), pp. 104–125. DOI: 10.1007/s00446-006-0155-x. URL: https://doi.org/10.1007/s00446-006-0155-x.

[56] Leslie Lamport. “Lower bounds on consensus”. In: *Unpublished manuscript* (2000).

[57] Andrew Lewis-Pye and Tim Roughgarden. "Byzantine Generals in the Permissionless Setting". In: *Financial Cryptography and Data Security - 27th International Conference, FC 2023, Bol, Brač, Croatia, May 1-5, 2023, Revised Selected Papers, Part I*. Ed. by Foteini Baldimtsi and Christian Cachin. Vol. 13950. Lecture Notes in Computer Science. Springer, 2023, pp. 21–37. DOI: 10.1007/978-3-031-47754-6_2. URL: https://doi.org/10.1007/978-3-031-47754-6%5C_2.

[58] Andrew Lewis-Pye and Tim Roughgarden. "Permissionless Consensus". In: *CoRR* abs/2304.14701 (2023). DOI: 10.48550/ARXIV.2304.14701. arXiv: 2304.14701. URL: https://doi.org/10.48550/arXiv.2304.14701.

[59] Andrew Lewis-Pye and Tim Roughgarden. "Permissionless consensus". In: *arXiv preprint arXiv:2304.14701* (2023).

[60] Yuan Lu et al. "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited". In: 2020, pp. 129–138.

[61] Marko Luksa. *Kubernetes in action*. Simon and Schuster, 2017.

[62] Nancy Lynch. "A hundred impossibility proofs for distributed computing". In: *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*. 1989, pp. 1–28.

[63] Nancy A. Lynch. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN: 9780080504704.

[64] Dahlia Malkhi, Atsuki Momose, and Ling Ren. "Byzantine Consensus under Fully Fluctuating Participation". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1448. URL: https://eprint.iacr.org/archive/2022/1448/20221024:011919.

[65] Dahlia Malkhi, Atsuki Momose, and Ling Ren. "Towards Practical Sleepy BFT". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 490–503.

[66] Dahlia Malkhi et al. "Objects shared by Byzantine processes". In: *Distributed Computing* 16.1 (2003), pp. 37–48.

[67] Andrew Miller et al. "The Honey Badger of BFT Protocols". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 31–42. ISBN: 9781450341394. DOI: 10.1145/2976749.2978399. URL: https://doi.org/10.1145/2976749.2978399.

[68] Atsuki Momose and Ling Ren. "Constant latency in sleepy consensus". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 2295–2308.

[69] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. "Signature-free asynchronous binary Byzantine consensus with t< n/3, O (n2) messages, and O (1) expected time". In: 62.4 (2015), pp. 1–21.

[70] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. "Signature-free asynchronous Byzantine consensus with t< n/3 and O (n2) messages". In: *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 2014, pp. 2–9.

[71] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Accessed: 2015-07-01. Dec. 2008. URL: `https://bitcoin.org/bitcoin.pdf`.

[72] Rafael Pass and Elaine Shi. "The sleepy model of consensus". In: *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II 23*. Springer. 2017, pp. 380–409.

[73] Michael O Rabin. "Randomized byzantine generals". In: *24th annual symposium on foundations of computer science (sfcs 1983)*. IEEE. 1983, pp. 403–409.

[74] Victor Shoup. "Practical threshold signatures". In: Springer. 2000, pp. 207–220.

[75] Swaminathan Sivasubramanian. "Amazon dynamoDB: a seamlessly scalable non-relational database service". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 729–730.

[76] Adriana Szekeres et al. "Meerkat: Multicore-scalable replicated transactions following the zero-coordination principle". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–14.

[77] Rebecca Taft et al. "Cockroachdb: The resilient geo-distributed sql database". In: *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2020, pp. 1493–1509.

[78] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. "Blockchain scaling using rollups: A comprehensive survey". In: *IEEE Access* 10 (2022), pp. 93039–93054.

[79] Pierre Tholoniat and Vincent Gramoli. "Formal verification of blockchain Byzantine fault tolerance". In: *arXiv preprint arXiv:1909.07453* (2019).

[80] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the tenth european conference on computer systems*. 2015, pp. 1–17.

[81] Cheng Wang et al. "Apus: Fast and scalable paxos on rdma". In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 94–107.

[82] Yang Xiao et al. "A survey of distributed consensus protocols for blockchain networks". In: *IEEE Communications Surveys & Tutorials* 22.2 (2020), pp. 1432–1465.

[83] Anatoly Yakovenko. *Solana: A new architecture for a high performance blockchain v0. 8.13*. 2018.

[84] Maofan Yin et al. "HotStuff: BFT consensus with linearity and responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 347–356.

[85] Guangsheng Yu et al. "Survey: Sharding in blockchains". In: *IEEE Access* 8 (2020), pp. 14155–14181.

[86] Haifeng Yu et al. "OHIE: Blockchain Scaling Made Simple". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 90–105. DOI: 10.1109/SP40000.2020.00008.

[87] Moti Yung. "The "Mobile Adversary" Paradigm in Distributed Computation and Systems". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. PODC '15. Donostia-San Sebastián, Spain: Association for Computing Machinery, 2015, pp. 171–172. ISBN: 9781450336178. DOI: 10.1145/2767386.2767453. URL: https://doi.org/10.1145/2767386.2767453.

[88] Irene Zhang et al. "Building consistent transactions with inconsistent replication". In: *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), pp. 1–37.