Distributed computing for blockchains and beyond

Andrei Tonkikh

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2024IPPAT041

Thèse de doctorat

Distributed Computing for Blockchains and Beyond

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Mathématiques et Informatique

Thèse présentée et soutenue à Palaiseau, le 16/12/2024, par

ANDREI TONKIKH

Composition du Jury :

Sebastien Tixeuil Professor, Sorbonne Université (LIP6)	Président/Examinateur
Maria Potop-Butucaru Professor, Sorbonne Université (LIP6)	Rapporteur
Vincent Gramoli Associate Professor, University of Sydney	Rapporteur
Sara Tucci-Piergiovanni Head of Laboratory, CEA List	Examinateur
Maurice Herlihy Professor, Brown University	Examinateur
Petr Kuznetsov Professor, Télécom Paris (LTCI)	Directeur de thèse

To my parents,
Elena and Alexander,
and my grandparents,
Ludmila and Alexander,
for their unconditional love and support
and for always believing in me

Acknowledgments

I would like to express my deepest gratitude to all the people who accompanied me throughout this journey.

First of all, to my supervisor, Prof. Petr Kuznetsov, who not only introduced me to the world of science but also served as a role model, showing by example what it means to be a kind and thoughtful person.

To all the people I had the pleasure of collaborating with, for the countless hours spent brainstorming in front of a whiteboard, on video calls, and in lengthy email exchanges. It is the excitement of sharing ideas with brilliant people like you that continually fueled my drive to push forward.

To my fellow PhD students and friends, Luciano and João—who shared with me the entire journey as we started and finished our PhD studies at around the same time—for all the fun we had together.

To my family, friends, and my amazing girlfriend Katya, who were there for me through the more difficult moments.

To the jury members and the reviewers who took the time to read this text and provide their invaluable feedback.

And, finally, to the distributed computing and blockchain communities, for being welcoming to newcomers and for innovating at a pace that continually renews my excitement for the field.

Abstract

In this dissertation, we address three major challenges in the design of blockchain systems in particular and large-scale fault-tolerant distributed systems in general. This work aims at improving the performance of such systems directly, as well as providing useful tools for future development of distributed algorithms.

First, we explore the limits of what can be done with minimal synchronization by designing CryptoConcurrency—an *asset transfer* system that, instead of totally ordering all users' requests, processes concurrent requests in parallel *as much as possible*. Unlike other similar systems, in CryptoConcurrency, we allow the users to have shared accounts and do not make the unrealistic assumption that an honest user's account is never accessed from two devices concurrently. CryptoConcurrency explores novel theoretical grounds by addressing transaction conflicts in a dynamic and non-pairwise manner, allowing the owners of each account to independently choose their preferred mechanism for conflict resolution.

Then, we improve the performance of *consensus*—the synchronization problem at the heart of most practical distributed systems. We build the first consensus protocol that manages to combine two desirable properties: extremely fast termination in favorable conditions and graceful recovery when such conditions are not met. The design involves a novel type of cryptographic proofs, with an efficient practical implementation.

Finally, we set out to tackle the problem of designing efficient distributed protocols with *weighted participation*. To this end, we define several new optimization problems, related to reducing or, in other words, quantizing the weights of the participants in a way that preserves important structural properties. We show how to apply them to make weighted-model variants of a large class of distributed protocols with very little overhead compared to their counterparts in the simpler non-weighted model. For these optimization problems, we prove upper bounds, provide a practical open-source approximate solver that satisfies these upper bounds, and perform an empirical study on the weight distributions from real-world blockchain systems.

Résumé

Dans cette thèse, nous abordons trois défis majeurs dans la conception des systèmes de blockchain en particulier et des systèmes distribués tolérants aux pannes à grande échelle en général. Ce travail vise à améliorer directement la performance de tels systèmes, ainsi qu'à fournir des outils utiles pour le développement futur d'algorithmes distribués.

Premièrement, nous explorons les limites de ce qui peut être réalisé avec une synchronisation minimale en concevant CryptoConcurrency—un système de *transfert d'actifs* qui, au lieu d'ordonner totalement toutes les requêtes des utilisateurs, traite les requêtes concurrentes en parallèle *autant que possible*. Contrairement à d'autres systèmes similaires, dans CryptoConcurrency, nous permettons aux utilisateurs d'avoir des comptes partagés et ne faisons pas l'hypothèse irréaliste qu'un compte d'utilisateur honnête n'est jamais accédé simultanément depuis deux dispositifs. CryptoConcurrency explore de nouveaux terrains théoriques en abordant les conflits de transactions de manière dynamique et non par paires, permettant aux propriétaires de chaque compte de choisir indépendamment leur mécanisme préféré de résolution de conflits.

Ensuite, nous améliorons la performance du *consensus*—le problème de synchronisation au cœur de la plupart des systèmes distribués pratiques. Nous construisons le premier protocole de consensus qui parvient à combiner deux propriétés souhaitables : une terminaison extrêmement rapide dans des conditions favorables et une récupération gracieuse lorsque ces conditions ne sont pas remplies. La conception implique un nouveau type de preuves cryptographiques, avec une implémentation pratique et efficace.

Enfin, nous nous attaquons au problème de la conception de protocoles distribués efficaces avec une *participation pondérée*. À cette fin, nous définissons plusieurs nouveaux problèmes d'optimisation, liés à la réduction ou, en d'autres termes, à la quantification des poids des participants d'une manière qui préserve d'importantes propriétés structurelles. Nous montrons comment les appliquer pour créer des variantes pondérées d'un large éventail de protocoles distribués avec très peu de surcharge par rapport à leurs homologues dans le modèle non pondéré plus simple. Pour ces problèmes d'optimisation, nous prouvons des bornes supérieures, fournissons un solveur pratique open-source approximatif qui satisfait ces bornes, et effectuons une étude empirique sur les distributions de poids provenant de systèmes de blockchain réels.

Résumé long en français

Contexte

Tolérance aux pannes byzantines dans les blockchains

Un système distribué est un ensemble d'acteurs, que nous appellerons *participants* ou *parties*, qui communiquent via un certain type de support (tel qu'Internet) et tentent d'atteindre un objectif commun. Un des principes centraux dans la conception de systèmes distribués est la tolérance aux pannes : la capacité d'un système à fonctionner comme prévu malgré la défaillance de certains de ses participants. Le type de défaillance le plus général parmi ceux couramment considérés est appelé faute *arbitraire* ou *byzantine* [101]. Comme son nom l'indique, il correspond à une situation où l'on fait peu ou pas d'hypothèses quant au comportement des parties défaillantes. Pour formaliser cette notion, l'approche standard consiste à considérer que toutes les parties défaillantes sont entièrement contrôlées par un même *adversaire*, dont l'unique objectif est de compromettre le système. C'est pourquoi ce type de défaillance est aussi appelé *faute malveillante*.

Bien qu'à l'origine on ait surtout eu à l'esprit des pannes matérielles et des bogues logiciels, l'utilisation la plus répandue aujourd'hui de la tolérance aux pannes byzantines (ou simplement BFT) se trouve dans les protocoles de blockchain. Au sens large, une *blockchain* est un système qui fonctionne sur un ensemble d'ordinateurs gérés par différentes personnes qui ne se font pas entièrement confiance, voire ne se connaissent pas. En fait, dans la plupart des blockchains publiques, n'importe qui peut devenir participant tout en restant anonyme, sous réserve d'investir un certain montant de ressources de calcul (dans les systèmes dits de *proof-of-work* [136]) ou de capital (dans les systèmes de *proof-of-stake* [135]). Ces systèmes gèrent souvent des actifs de très grande valeur monétaire, avec une capitalisation de marché totale estimée à plusieurs billions de dollars pour l'ensemble des actifs de blockchains publiques au moment de la rédaction. Étant donné les enjeux considérables, il est quasiment inévitable que des acteurs malveillants tentent de s'infiltrer dans le système et de le manipuler pour en extraire de la valeur ou, tout simplement, de le briser dans le but de nuire aux autres.

Cependant, les protocoles traditionnels tolérants aux pannes byzantines ne suffisent pas à satisfaire les besoins des blockchains modernes. Ils ont été conçus dans l'optique d'un petit nombre de parties, généralement inférieur à dix, tandis que dans les blockchains on peut compter des centaines, voire des dizaines de milliers de participants. De plus, la vision aujourd'hui popularisée du « web-3 » suggère que, à terme, la plupart des services Internet utiliseront des blockchains pour représenter la propriété numérique, ce qui ne sera possible que si les blockchains parviennent à proposer des latences très faibles combinées à un débit extrêmement élevé. Tout cela nécessite une nouvelle génération de protocoles, qui s'appuient souvent sur une cryptographie sophistiquée.

Mais qu'est-ce qu'une blockchain et pourquoi devrais-je m'y intéresser ?

Les blockchains, au sens moderne du terme, ont vu le jour à la suite d'un article pseudonyme décrivant Bitcoin [112]. L'objectif annoncé par les auteurs était de créer une monnaie numérique et un système de paiement fonctionnant avec cette monnaie, qui ne seraient possédés ni contrôlés par aucune personne, organisation ou pays, mais qui émergeraient plutôt du comportement collectif de nombreux participants indépendants. Ces participants, appelés mineurs dans l'article sur Bitcoin, investissent leurs ressources pour entretenir le système et reçoivent en retour une récompense sous forme de la même monnaie numérique. Cette application à elle seule est déjà assez puissante. Une monnaie émise sur la base d'un ensemble de règles publiques et prédéfinies peut constituer une alternative attrayante (ou peut-être

un complément précieux) au système monétaire actuel, où la création monétaire est décidée à volonté par les gouvernements. D'autre part, un système de transfert d'actifs qui n'est pas fondé sur des relations de confiance complexes entre banques permet des transferts internationaux de valeur beaucoup plus rapides et moins coûteux.

Cependant, l'aspect potentiellement encore plus intéressant réside dans le mécanisme sous-jacent sur lequel ce système de paiement est construit : en essence, une base de données distribuée, décentralisée, et jusqu'à un certain point programmable, qui maintient un registre de la « propriété » des BTC — la monnaie — et facilite son transfert. Par la suite, à la suite du succès de Bitcoin, une multitude d'autres systèmes similaires sont apparus [134]. Le plus notable est sans doute Ethereum [139], qui s'est alors concentré principalement sur l'enrichissement des capacités de programmation de la base de données sous-jacente.

Une bonne manière abstraite de penser à une *blockchain publique* est d'imaginer un ordinateur magique quelque part dans le « nuage », qui ne tombe jamais en panne ni ne commet d'erreurs, auquel tout le monde a accès (selon un ensemble de règles d'interface données), mais que personne ne possède ni ne contrôle. Qu'on adhère ou non à l'utilité de chaque application en particulier, il est clair que c'est une capacité nouvelle qui n'existait pas auparavant. Dans un monde sans blockchain, toute base de données doit être gérée par une entité, qu'il s'agisse d'une administration publique ou d'une entreprise. Dans ce monde, toutes les interactions numériques à caractère public, commerciales ou autres, passent par une instance centrale, qui est en général une société commerciale. En outre, au sein de cette société, des personnes physiques disposent d'un accès administrateur direct à la base de données.[1] Cela suscite des problèmes d'alignement des intérêts, de maîtrise des coûts et de sécurité, tout en compliquant les interactions transfrontalières lorsque la recherche d'une source de confiance commune s'avère problématique.

Les *blockchains privées*, quant à elles, sont souvent contrôlées par une seule organisation, mais renforcent la sécurité en rendant nécessaire l'accès à plusieurs serveurs pour toute tentative de falsification de la base de données. Enfin, les blockchains dites *de consortium* permettent à plusieurs organisations d'établir un socle de confiance commun, indispensable à une collaboration efficace, chacune contrôlant un sous-ensemble de participants du protocole blockchain, de sorte qu'aucune organisation ne dispose d'un accès privilégié à la base de données partagée.

Malgré tout ce potentiel, de nombreux obstacles se dressent encore pour une adoption à grande échelle, et leur analyse détaillée sort du cadre de ce travail. Cependant, même aujourd'hui, seize ans après la publication initiale de l'article sur Bitcoin, l'un des principaux freins reste la technologie. Les protocoles actuels ne sont tout simplement pas prêts à supporter la charge nécessaire tout en offrant une latence suffisamment faible et en passant à l'échelle pour accueillir un grand nombre de participants, condition essentielle pour assurer la décentralisation du système. Dans cette thèse, nous entendons contribuer à la fois à une meilleure compréhension théorique de la tolérance aux pannes byzantines et à l'amélioration technologique des blockchains.

Notre contribution

Au cœur même des blockchains modernes se trouve sans doute le problème le plus étudié de l'informatique distribuée : le *consensus* [63]. Plus précisément, la plupart des blockchains implémentent une variante du consensus connue dans la littérature de l'informatique distribuée traditionnelle sous le nom de *réplication de machines à états (SMR)* [96]. Ce concept formalise l'idée d'un « ordinateur magique quelque part dans le nuage qui ne tombe jamais en panne ni ne commet d'erreurs ». Le défi consiste à construire cet ordinateur à partir d'un ensemble de machines bien réelles et susceptibles de dysfonctionner, voire d'être contrôlées de manière malveillante. C'est un problème très difficile même lorsque seules des pannes de type « crash » sont envisagées, et il l'est d'autant plus dans le monde des pannes byzantines [10, 55, 56, 58, 63].

Chapitre 2 s'appuie sur un travail conjoint avec Pavel Ponomarev, Petr Kuznetsov et Yvonne-Anne Pignolet [133], dans lequel nous contournons le consensus pour résoudre le problème de la création d'un système de paiement (ou, plus généralement, un *système de transfert d'actifs*) directement à partir de principes de base. Contrairement à des travaux antérieurs similaires [19, 48, 78, 79, 92, 126], nous évitons l'hypothèse, trop simpliste, selon laquelle chaque compte appartiendrait à un unique propriétaire et qu'aucun compte d'utilisateur honnête ne pourrait jamais être utilisé de façon concurrente (par exemple,

[1] Sauf si l'entreprise exploite une *blockchain privée*, comme évoqué ci-dessous.

depuis plusieurs appareils). Bien qu'il soit fondamentalement impossible de se passer entièrement du consensus sans une telle hypothèse [78], nous réussissons à l'éviter sur le chemin critique et ne nous appuyons que sur une forme affaiblie de consensus, que nous avons appelée « consensus de compte », laquelle vise essentiellement à permettre aux propriétaires d'un compte partagé de se mettre d'accord entre eux. De plus, nous ne la sollicitons que dans le cas rare où des transactions émises concurremment dépassent le solde du compte. Sur le plan formel, il s'agit bien d'une forme de consensus, mais cela n'a rien à voir avec la nécessité de parvenir à un accord entre *tous* les participants du système *à chaque fois* que quelqu'un souhaite effectuer un transfert. Avec ce travail, nous espérons contribuer à la fois à la conception de systèmes pratiques et à la compréhension théorique fondamentale des types de problèmes qui peuvent se résoudre sans recourir à l'arsenal lourd du consensus.

Bien que le transfert d'actifs reste l'application la plus courante des blockchains, il existe de nombreuses autres applications importantes nécessitant un consensus, comme les contrats intelligents génériques [137]. Ainsi, même avec un système de transfert d'actifs plus efficace, le consensus demeure indispensable. Il est essentiel d'en améliorer les performances tout en maintenant la capacité du protocole à s'adapter à un grand nombre de participants, afin de préserver une décentralisation suffisante.

Chapitre 3 est basé sur un travail conjoint avec Matthieu Rambaud et Mark Abspoel [122], dans lequel nous montrons comment accélérer le consensus à l'aide d'une *voie rapide*, assurant la terminaison en seulement quelques délais de messages dans des conditions favorables, tout en maintenant la *complexité en authenticateurs* (une mesure qui combine la charge de communication et de calcul dans le système) linéaire en fonction du nombre de participants. Avant nos travaux, les constructions similaires de voies rapides [74, 90, 99] s'accompagnaient d'une complexité quadratique en cas de défaillance du leader ou de retards réseau inattendus, ce qui les rendait inadaptées à des déploiements à grande échelle. Pour obtenir ce résultat, nous introduisons un nouveau type de preuve cryptographique, que nous appelons *Proof-of-Exclusivity (PoE)*, et en proposons une implémentation efficace à partir de signatures de seuil. De plus, nous utilisons l'*responsabilisation* (accountability) et la réutilisation de signatures afin que la construction d'une PoE n'induise *aucun surcoût* tant qu'aucun participant n'équivaut ouvertement. En intégrant un mécanisme de reconfiguration [100] ou de *slashing* [33], il est possible de sanctionner tout participant malhonnête, rendant ce cas extrêmement improbable en pratique.

Depuis la publication de [122], notre construction PoE, appelée *Big Buckets PoE*, a trouvé une application indépendante dans l'amélioration de la complexité adaptative du consensus synchrone [46].

Ces deux chapitres suivent la convention quasiment universelle de la littérature sur la tolérance aux pannes byzantines : on considère un système de n nœuds dont jusqu'à f peuvent être byzantins, avec f limité à une fraction de n (les cas les plus courants étant $f < n/3$ ou $f < n/2$). Ironiquement, malgré le fait qu'il soit omniprésent dans la littérature et l'analyse formelle des protocoles, ce modèle est trop simplificateur et ne représente pas les systèmes réels. En pratique, les blockchains publiques fonctionnent généralement dans un modèle *pondéré*, où chaque participant est associé à un *poids* et où l'adversaire peut corrompre un ensemble de parties détenant au plus une fraction (par exemple, $1/3$ ou $1/2$) du *poids total*. Si certains protocoles se traduisent relativement facilement dans ce nouveau cadre au moyen de « votes pondérés », d'autres s'appuient sur des composants qui semblent intrinsèquement discrets, comme le partage de secrets [123] ou les codes correcteurs et effaçables [105].

Nous abordons ce problème dans le Chapitre 4, qui s'appuie sur un travail conjoint avec Luciano Freitas [132]. Nous formalisons et proposons des solutions approchées à trois nouveaux problèmes d'optimisation, que nous regroupons sous le nom générique de *problèmes de réduction de poids*.[2] Fait quelque peu surprenant, nous découvrons qu'il est possible de projeter efficacement de grands poids réels en de petits poids entiers, tout en préservant certaines propriétés structurelles essentielles. Nous montrons ensuite comment appliquer ces problèmes de réduction de poids pour transformer une vaste classe de protocoles conçus dans le modèle classique (ou, selon notre terminologie, *nominal*) vers le modèle pondéré. Dans certains cas, notre transformation ne nécessite qu'une très faible baisse de la résilience, mais, de manière assez remarquable, pour de nombreux problèmes importants, nous parvenons à proposer des solutions pondérées offrant la même résilience que celles du modèle nominal, grâce à une utilisation sélective et soignée de la réduction de poids. Parmi ces exemples notables, on trouve le stockage distribué à codes effaçables, la diffusion authentifiée (broadcast), le partage de secrets vérifiable ou encore le consensus asynchrone. Nous pensons que ce travail est une contribution importante non seulement pour l'informatique distribuée, mais aussi pour la cryptographie appliquée, car il apporte une solution pratique à une difficulté de longue date : l'adaptation de la cryptographie de seuil au modèle pondéré.

[2] Un nom plus descriptif, quoique moins précis, pourrait être « quantification des poids ».

La réduction de poids a déjà trouvé une application dans l'implémentation de balises aléatoires (random beacons) pour les blockchains en proof-of-stake [140].

Publications

Les chapitres techniques de cette dissertation s'appuient sur trois articles publiés :

- Andrei Tonkikh, Pavel Ponomarev, Petr Kuznetsov, and Yvonne-Anne Pignolet. Cryptoconcurrency:(almost) consensusless asset transfer with shared accounts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1556–1570, 2023.
- Matthieu Rambaud, Andrei Tonkikh, and Mark Abspoel. Linear view change in optimistically fast bft. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, pages 67–78, 2022.
- Andrei Tonkikh and Luciano Freitas. Swiper: a new paradigm for efficient weighted distributed protocols. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 283–294, 2024.

L'auteur a également participé à la publication de plusieurs autres articles qui ne figurent pas dans cette dissertation :

- Luciano Freitas, Andrei Tonkikh, Adda-Akram Bendoukha, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, and Petr Kuznetsov. Homomorphic sortition – single secret leader election for PoS blockchains. Cryptology ePrint Archive, Paper 2023/113, 2023. URL: `https://eprint.iacr.org/2023/113`
- Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer. *Distributed Computing*, 36(3):349–371, 2023
- Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Brief announcement: Asynchronous randomness and consensus without trusted setup. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 103–105, 2022
- Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Distributed randomness from approximate agreement. In *36th International Symposium on Distributed Computing*, 2022
- Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. *Distributed Computing*, 35(6):477–502, 2022
- Luciano Freitas de Souza, Andrei Tonkikh, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov. Randsolomon: Optimally resilient random number generator with deterministic termination. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022
- Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast byzantine consensus. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 343–353, 2021
- Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021
- Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 26–38. IEEE, 2020

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 Byzantine fault tolerance in blockchains

A distributed system is one that consists of a number of actors, which we will refer to as *participants* or *parties*, communicating over some kind of medium (such as the Internet) and trying to achieve a common objective. One of the central principles in the design of distributed systems is fault tolerance: the ability of the system to function as intended despite the failures of some of the participants. The most general of the commonly considered types of failures is called *arbitrary* or *Byzantine* [101] faults. As the name suggests, it corresponds to making no or very few assumptions on the behavior of faulty parties. To formalize this notion, the standard approach is to consider all the faulty parties as being under the total control of a single *adversary*, whose sole goal is to break the system. Hence, yet another name for this type of failures is *malicious* faults.

While originally conceived with hardware malfunctions and software bugs in mind, the prevalent use of Byzantine fault tolerance (or simply BFT) today is in blockchain protocols. In the general sense, a *blockchain* is a system that runs on a set of computers managed by different people who do not fully trust or even know each other. In fact, in the case of most public blockchains, anyone can become a participant while staying anonymous, provided they invest a certain amount of computational resources (in the so-called *proof-of-work* systems [136]) or capital (in *proof-of-stake* systems [135]). These systems often manage assets of very high monetary value, with the total market capitalization of all public blockchain assets estimated to be in the trillions of dollars at the time of writing. With stakes so high, it is all but inevitable that malicious actors will try to infiltrate the system and either manipulate it to extract value for themselves or simply break it to harm others.

However, traditional Byzantine fault-tolerant protocols are not sufficient to satiate the needs of modern blockchains. They were designed with a small number of parties in mind, typically below ten, while the number of participants in blockchain systems ranges from hundreds to tens of thousands. Moreover, the newly popularized vision of "web-3" suggests that eventually most Internet services will use blockchains to represent digital ownership, which is only ever going to be possible if blockchains manage to achieve very low latencies combined with extremely high throughput. All of this necessitates a new set of protocols, often relying on sophisticated cryptography for their construction.

1.1.2 But what is a blockchain and why should I care?

Blockchains, in the modern sense of the word, started from a pseudonymous paper introducing Bitcoin [112]. The authors' stated goal was to create a digital currency and a payment system operating in that currency that would not be owned or controlled by any person, organization, or country, but instead emerged from the collective behavior of many individual participants. The participants, called miners in the Bitcoin paper, would invest their resources into maintaining the system and in return would receive a reward in the form of that same digital currency. This application alone is quite powerful. A currency that is issued purely based on a set of publicly known pre-set rules may provide an attractive alternative (or, perhaps, a valuable complement) to the current monetary system, where money is created at will

by governments. On the other hand, the asset transfer system that is not based on complicated trust relationships between banks allows for much faster and cheaper international value transfers.

However, what is potentially even more interesting is the underlying mechanism that this payment system was built upon, which is essentially a distributed, decentralized, and, to a limited degree, programmable database maintaining the records of "ownership" of BTC—the currency—and facilitating its transfers. Later, following Bitcoin's success, a myriad of other similar systems emerged [134]. Most notably, Ethereum [139], which, at the time, focused primarily on enriching the programmability of the underlying database.

A good abstract way of thinking about a *public* blockchain system is to imagine a magical computer somewhere in the sky that never crashes or errs, that everyone has access to (within a fixed set of interfacing rules), but no one owns or controls. While the utility of specific applications is still a subject for debate, it is easy to see that this is a new capability that did not exist before. In a pre-blockchain world, every database must be managed by some entity, be it a government agency or an organization. In that world, all digital public interactions, commercial or otherwise, must go through some central party, typically a for-profit company. Moreover, within that company, there are individual people with direct admin access to the database.[1] This creates incentive alignment, cost efficiency, and security issues and complicates cross-border interactions where finding common anchors of trust may be problematic.

On the other hand, *private* blockchains are often controlled by a single organization, but provide additional security by making sure that any tampering with the database requires the attacker to obtain access to multiple servers. Finally, *consortium* blockchains allow multiple organizations establish a common pivot of trust necessary for efficient collaboration by each controlling a subset of participants in a blockchain protocol, thus ensuring that no single organization has privileged access to the shared database.

Despite all of that potential, the barriers for its full realization are numerous and their detailed analysis is outside the scope of this work. However, even today, 16 years after the initial publication of the Bitcoin paper, at least one of the major barriers is still technological. The current protocols are simply not ready to handle the necessary load while simultaneously providing sufficiently low latency and scaling to enough participants to ensure that the system is sufficiently decentralized. In this dissertation, we attempt to contribute to both the theoretical understanding of Byzantine fault tolerance and the technological development of blockchains.

1.2 Our contribution

At the very heart of modern blockchain systems lies what is, perhaps, the most widely studied problem of distributed computing—*consensus* [63]. More precisely, most blockchains implement a variant of consensus known in traditional distributed computing literature as *state machine replication (SMR)* [96]. It formally captures the abstraction of a "magical computer somewhere in the sky that never crashes or errs". The challenge, however, is in constructing it out of a number of non-magical and error-prone or even sometimes maliciously controlled computers. This is a very challenging problem even when simple crash failures are concerned and even more so in the world with Byzantine faults [10, 55, 56, 58, 63].

Chapter 2 is based on a joint work with Pavel Ponomarev, Petr Kuznetsov, and Yvonne-Anne Pignolet [133], in which we sidestep consensus and solve the problem of creating a payment system (or, more generally, an *asset transfer* system) directly, from the first principles. Unlike prior similar works [19, 48, 78, 79, 92, 126], we avoid making the overly simplistic assumption that every account belongs to a single owner only and no account belonging to an honest user can ever be accessed concurrently (e.g., from multiple devices). Although it is fundamentally impossible to completely avoid consensus without such an assumption [78], we manage to avoid it on the critical path and only use a certain weak form of consensus that we dubbed "account consensus", which essentially means that the owners of a shared account need to be able to reach an agreement between themselves. Furthermore, we only rely on it in the rare case when concurrently issued transactions exceed the account balance. While, from the formal point of view, it is a form of consensus, it is not the same as having to reach an agreement among *all* the participants of the system *each* time someone wants to make a transfer. With this work, we aim to contribute both to practical system design and to the fundamental theoretical

[1] Unless the company runs a *private* blockchain, as described below.

understanding of the kinds of problems that are possible to solve without involving the heavy machinery of consensus.

While asset transfer is still the most common application of blockchains, there are many other important applications that do require consensus, such as general-purpose smart contracts [137]. Hence, even with a separate, more efficient asset transfer system in place, consensus will remain necessary. It is essential to improve its performance while keeping the protocol scalable to allow as many independent parties as possible to participate, ensuring sufficient decentralization.

Chapter 3 is based on a joint work with Matthieu Rambaud and Mark Abspoel [122], in which we show how to speed up consensus with a *fast track*, ensuring termination in just a couple of message delays in favorable conditions, while keeping the authenticator complexity (a measure that combines communicational and computational load on the system) linear in the number of participants. Prior to our work, similar fast track constructions [74, 90, 99] incurred quadratic complexity in case of a faulty leader or unexpected network delays, making them unsuitable for large-scale deployments. To achieve this result, we introduce a novel type of a cryptographic proof that we dub a *Proof-of-Exclusivity (PoE)* and provide an efficient instantiation of such a proof from threshold signatures. Moreover, we employ *accountability* and reuse signatures to ensure that a PoE can be constructed with *zero overhead* unless a participant openly equivocates. With the help of a reconfiguration [100] or a slashing [33] mechanism, we can punish the misbehaving participant, thus making this case extremely unlikely in practice.

Since the publication of [122], our PoE construction, called *Big Buckets PoE*, has found an independent application in improving the adaptive complexity of synchronous consensus [46].

These two chapters follow what is a nearly universal standard for the Byzantine fault tolerance literature: they assume a system of n nodes of which up to f can be Byzantine, where f is limited to a fraction of n ($f < n/3$ or $f < n/2$ are the most common choices). Ironically, despite being ubiquitous in the literature and formal analysis of protocols, this model is oversimplified and does not represent real systems. Instead, public blockchains typically operate in a *weighted* model, where each participant is associated with a *weight* and the adversary can corrupt a set of parties holding at most a fraction (e.g., $1/3$ or $1/2$) of the *total weight*. While some protocols are relatively easy to convert to such a model with what we call "weighted voting", others rely on components that seem to be inherently discrete in nature, such as secret sharing [123] or erasure and error-correcting codes [105].

We address this issue in Chapter 4, which is based on a joint work with Luciano Freitas [132]. We formalize and provide approximate solutions to three novel optimization problems, which we collectively call *the weight reduction problems*.[2] The somewhat surprising discovery is that it is possible to efficiently map large real weights into small integer weights while preserving certain critical structural properties. We then proceed to demonstrate how to apply the weight reduction problems to transform a large class of protocols designed in the classical (or, how we call it, *nominal* model) to the weighted model. While, for some protocols, our transformation requires an arbitrarily small reduction in resilience, surprisingly, for many important problems, we manage to obtain weighted solutions with the same resilience as nominal ones through a more careful selective application of weight reduction. Notable examples include erasure-coded distributed storage and broadcast protocols, verifiable secret sharing, and asynchronous consensus. We believe that this work provides a valuable contribution not only to distributed computing, but also to applied cryptography as it provides a practical solution to the long-standing challenge of adapting threshold cryptography to the weighted model.

Weight reduction has already found an application in the implementation of random beacons in proof-of-stake blockchains [140].

1.3 Publications

The technical chapters in this dissertation are based on three published articles:

- Andrei Tonkikh, Pavel Ponomarev, Petr Kuznetsov, and Yvonne-Anne Pignolet. Cryptoconcurrency:(almost) consensusless asset transfer with shared accounts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1556–1570, 2023.

- Matthieu Rambaud, Andrei Tonkikh, and Mark Abspoel. Linear view change in optimistically fast bft. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, pages 67–78, 2022.

[2] Perhaps, a more descriptive albeit slightly less precise name could be "weight quantization".

- Andrei Tonkikh and Luciano Freitas. Swiper: a new paradigm for efficient weighted distributed protocols. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 283–294, 2024.

The author also participated in the publication of several other articles that were not included in this dissertation:

- Luciano Freitas, Andrei Tonkikh, Adda-Akram Bendoukha, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, and Petr Kuznetsov. Homomorphic sortition – single secret leader election for PoS blockchains. Cryptology ePrint Archive, Paper 2023/113, 2023. URL: `https://eprint.iacr.org/2023/113`
- Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer. *Distributed Computing*, 36(3):349–371, 2023
- Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Brief announcement: Asynchronous randomness and consensus without trusted setup. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 103–105, 2022
- Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Distributed randomness from approximate agreement. In *36th International Symposium on Distributed Computing*, 2022
- Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. *Distributed Computing*, 35(6):477–502, 2022
- Luciano Freitas de Souza, Andrei Tonkikh, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov. Randsolomon: Optimally resilient random number generator with deterministic termination. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022
- Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast byzantine consensus. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 343–353, 2021
- Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021
- Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 26–38. IEEE, 2020

1.4 Roadmap

Chapter 2 presents our asset transfer protocol with minimal synchronization. Chapter 3 describes a construction of a Byzantine fault-tolerant consensus protocol with fast track and linear complexity. Chapter 4 addresses the problem of converting distributed protocols to the weighted model. Each chapter concludes with a discussion of interesting directions for future research, related to each of the topics (Sections 2.14, 3.9 and 4.12). We conclude the dissertation in Chapter 5 with a brief discussion of more general topics of interest in the fields of Byzantine fault-tolerant distributed computing and blockchains.

1.5 Funding

This thesis was supported by Mazars Group and the TrustShare Innovation Chair project.

Chapter 2

(Almost) Consensusless Asset Transfer with Shared Accounts

2.1 Introduction

The ability to transfer assets from one user's account to another user's account despite the potential presence of malicious parties comes naturally when the users are able to solve Byzantine fault-tolerant consensus [101] in order to reach an agreement on the evolution of the system state. They can simply agree on the order in which their transactions are executed. Indeed, for a long time consensus-based blockchain protocols [112, 139] have remained the *de facto* standard to implement *asset transfer* (also known as *cryptocurrency*).

However, Byzantine fault-tolerant consensus is a notoriously hard synchronization problem. Not only is it impossible to solve deterministically in asynchronous systems [63], there are also harsh lower bounds on its costs even with stronger synchrony assumptions: at least $\Omega(f^2)$ messages [55] and $\Omega(f)$ rounds of communication [10, 56, 58], even in the synchronous model (i.e., when there is a known upper bound Δ on the time it takes for a message sent by a correct process to reach its destination). Despite the efforts of many brilliant researchers and engineers, existing consensus-based blockchain implementations still struggle to achieve latency, throughput, and transaction fees acceptable for widely applicable payment systems.

The good news is that consensus is not necessary to implement an asset transfer system [78, 79]. This observation led to a series of purely asynchronous, *consensus-free* cryptocurrencies [19, 48, 92, 125]. Practical evaluations have confirmed that such solutions have significant advantages over consensus-based protocols in terms of scalability, performance, and robustness [19, 48].

However, all existing consensus-free implementations share certain limitations. In particular, they assume that each account is controlled by a single user that never issues multiple transactions in parallel. This assumption precludes sharing an account by multiple users, e.g., by family members, or safely accessing it from multiple devices. If an honest user accidentally issues several concurrent transactions, the existing implementations may block the account forever, without any possibility to recover it. Consensus-free systems based on the UTXO model [138], such as [125], share the same restriction.

In this chapter, we propose *CryptoConcurrency*, a hybrid protocol that combines the benefits of both approaches. It allows accounts to be shared by multiple users and avoids using consensus in most cases. Indeed, as demonstrated in [78], in certain cases, consensus is unavoidable. Therefore, the challenge is to minimize its use.

Informally, in our implementation, if transactions concurrently issued on the same account can *all* be applied without exhausting the account's balance, they are processed in parallel, in a purely asynchronous way (i.e., without invoking consensus). This property appears natural as such transactions can be ordered arbitrarily, and the order will not affect the resulting account's state. In contrast, when the account balance does not allow for accepting all of the concurrent transactions, the account owners may use consensus to agree which transactions should be accepted and which ones should be considered failed due to the lack of funds.

Our protocol dynamically detects the cases when consensus should be used. This distinguishes our

approach from earlier work on combining weak and strong synchronization in one implementation, where conflicts were defined in a *static* way, i.e., any *potentially* conflicting concurrent operations incur the use of consensus, both in general-purpose systems [20, 102, 104, 118] and in systems specialized for asset transfer [125].

Interestingly, every account can be associated with distinct consensus instances that only need to be accessed by the account's owners. In practice, consensus instances for different accounts can be implemented in different ways and on different hardware, depending on the trust assumptions of their owners.

We believe that the results of this chapter can be further generalized to applications beyond asset transfer and that this work can be a step towards devising *optimally-concurrent* protocols that dynamically determine the cases when falling back to stronger synchronization primitives is unavoidable. Intuitively, it seems to be possible for an object with a sequential specification [82] to operate in a purely asynchronous manner without resorting to consensus in any executions where reordering of the concurrent operations does not affect their outcomes. This enables the development of lightweight, adaptive implementations that can avoid the costs of heavy synchronization primitives in most cases without compromising functionality or sacrificing liveness even in highly concurrent scenarios.

Roadmap. The rest of this chapter is organized as follows. We overview related work in Section 2.2. In Section 2.3, we introduce our system model. In Section 2.4, we overview the basic principles of our algorithm. We state the problem and the main theorem of the chapter formally in Section 2.5, describe the key building blocks in Sections 2.6 and 2.7 and provide the complete protocol in Section 2.8. Details on the algorithm and its proof of correctness are delegated to Sections 2.10 to 2.13. We conclude the chapter with a discussion of the directions for future work in Section 2.14.

2.2 Related Work

protocol	**resilience**	**worst-case end-to-end latency** (in round-trip times)		
		no concurrency on the account	k **concurrent requests, no overspending**	k **concurrent requests, with overspending**
Consensus-based	$f < n/3$	Global Consensus	Global Consensus	Global Consensus
k**-shared AT [78]**	$f < n/3$	Account Consensus	Account Consensus	Account Consensus
Astro II [48] / FastPay [19]	$f < n/3$	**2** RTTs with $O(n)$ msgs	Not supported	Not supported
Consensus on Demand [125]	$f < n/5$	**2** RTTs with $O(n^2)$ msgs [1]	Global Consensus [2]	Global Consensus [2]
CryptoConcurrency	$f < n/3$	**5** RTTs with $O(n)$ msgs [3]	$k + 4$ RTTs	Account Consensus

1 The original paper does not consider how the client learns a relevant sequence number. Hence, we added one round-trip for the client to fetch it. Note that the local client's sequence number can be outdated unless the client is also required to act as a replica and to stay online observing all other transactions.

2 The original paper considers only Global Consensus, trusted by all parties. However, we believe that it is possible to make a version of [125] that relies only on Account Consensus without affecting latency in case of absence of concurrency, using techniques similar to those used in CryptoConcurrency.

3 2 out of 5 RTTs are used to fetch an up-to-date initial state. We discuss potential ways to avoid it as well as other directions for optimizations in Section 2.8.4.

Table 2.1: Asset transfer protocol comparison.

Conventionally, asset transfer systems (or cryptocurrencies) were considered to be primary applications of blockchains [13, 112, 139], consensus-based protocols implementing replicated state machines. In [78, 79], it has been observed that asset transfer *per se* does not in general require consensus. This observation gave rise to simpler, more efficient and more robust implementations than consensus-based solutions [19, 48, 92, 125]. These implementations, however, assume that no account can be concurrently debited, i.e., no conflicting transactions must ever be issued by honest account owners.

In this chapter, we propose an asset-transfer implementation in the setting where users can share an account and, thus, potentially issue conflicting transactions. Our implementation does resort to consensus

in some executions, which is, formally speaking, inevitable [78]. Indeed, as demonstrated in [78], a fully consensus-free solution would be impossible, as there is a reduction from consensus to asset transfer with shared accounts. The algorithm for asset transfer with account sharing presented in [78] uses consensus for all transfers issued by accounts owned by multiple clients, even if the account owners never try to access the account concurrently.

Lamport's Generalized Paxos [102] describes a state-machine replication algorithm for executing concurrently applied non-conflicting commands in a *fast way*, i.e., within two message delays. The algorithm involves reaching agreement on a partially ordered *command-structure* set with a well-defined least upper bound. The approach, however, cannot be applied directly in our case, as it assumes that every set of pairwise compatible operations can be executed concurrently [102, p. 11]. This is not the case with asset transfer systems: imagine three transactions operating on the same account, such that applying all three of them drain the account to a negative balance, but every two of them do not. In order to account for such transactions, we therefore have to further generalize Generalized Paxos, in addition to taking care of Byzantine faults. (The original protocol was designed for the crash-fault model, though an interesting Byzantine version has been recently proposed [118]).

Byblos [20], a "clairvoyant" state machine replication protocol, further improves upon Byzantine Generalized Paxos by making it leaderless and compatible with a more general class of consensus protocols for the fallback at the cost of sub-optimal resilience ($n \geq 4f + 1$). However, it also considers a static definition of conflicts.

In [104], RedBlue consistency was introduced. It manifests a different approach to combining weak (asynchronous) and strong (consensus-based) synchronization in one implementation. In defining the sequential specification of the object to be implemented, the operations are partitioned *a priori* into blue (parallelizable) and red (requiring consensus). In the case of asset transfer, transfer operations would be declared red, which would incur using consensus among all clients all the time.

Generalized Lattice Agreement [61] has emerged as a useful abstraction for achieving agreement on comparable outputs among clients without resorting to consensus. One can use it to build a fully asynchronous state-machine replication protocol assuming that *all* operations are commutative (can be executed in arbitrary order, without affecting the result). In this chapter, we further extend these ideas to operations that are not always commutative.

Recent work by Sliwinski, Vonlanthen, and Wattenhofer [125] aims to achieve the same goal of combining the consensus-free and consensus-based approaches as we do in this chapter. It describes an asset-transfer implementation that uses consensus whenever there are two concurrent transactions on the same account, regardless of the account's balance. The algorithm assumes $5f + 1$ replicas, where up to f can be Byzantine, and a central consensus mechanism trusted by all participants. In contrast, CryptoConcurrency implements dynamic (balance-based) overspending detection with the optimal number of $3f + 1$ replicas and without universally trusted consensus, but has higher latency in conflict-free executions. We achieve this by introducing a new abstraction that combines and then extends key ideas from Generalized Lattice Agreement [61] and Paxos [102] as will be further elaborated in Section 2.6.

We summarize the performance of CryptoConcurrency compared to similar protocols in Table 2.1. In the absence of concurrent transactions on the same account, the end-to-end latency of CryptoConcurrency is 5 round-trips, compared to 2 in [48] and [125]. If k concurrent transactions on the same account can all be satisfied without overspending, the worst-case latency will be $k + 4$ round-trips, whereas [125] would fall back to consensus, and [48] would lose liveness.

The higher latency of CryptoConcurrency is mainly due to the fact that we do not assume that clients have an up-to-date state when they start executing a transaction (in principle, a "client" can be simply a smart card storing a private key connected to a mobile point-of-sale device). Furthermore, since the main goal was to demonstrate the possibility rather than to achieve the best performance, we preferred simplicity over efficiency and opted for a highly consistent storage system (see Section 2.7 for details). Hence, we spend 2 round-trips to obtain the relevant state at the beginning of each operation. We provide a more detailed latency breakdown and discuss potential ways to decrease it in Section 2.8.4.

2.3 System Model

2.3.1 Processes, clients and replicas

Let Π be a (possibly infinite) set of potentially participating *processes*. We assume that there is a fixed subset of n processes Π, called *replicas*, that verify operations and maintain the *system state*. Every process can also act as a *client* that invokes *operations* on the shared state. We make no assumptions on the number of clients in the system or on their availability, i.e., a client can go offline between its requests to the system. In the definitions and proofs, we impose the standard assumption of the existence of a *global clock*, not accessible to the processes.

We assume an adaptive adversary that can corrupt any process at any moment in time. Once corrupted, the process falls under complete control of the adversary. For simplicity of presentation, we say that a process is *correct* if it is never corrupted during the whole execution and is *Byzantine* otherwise.

The adversary can perfectly coordinate all Byzantine processes, and it is aware of the entire system state at any point of time except for the private cryptographic keys of the correct processes. We rely on the standard assumption that the computational power of the adversary is bounded so that it cannot break cryptographic primitives, such as digital signatures.

We assume that any number of clients and $f < n/3$ replicas can be Byzantine and that each pair of correct processes can communicate over a *reliable authenticated channel*.

2.3.2 Accounts

We assume a set of *accounts* $\mathcal{A}$ across which assets are exchanged in the system. As every account can be owned by multiple processes, we equip accounts with a map $\mu : \mathcal{A} \to 2^{\Pi}$ that associates each account with a finite set of clients that can perform debit operations on it. We say that client $q \in \mu(a)$ is an owner of an account a (q owns account a). To simplify the model, we suppose that no client owns more than one account. To use two or more accounts, one is required to have multiple client instances.

Account a is called *correct* if it is owned by correct clients, i.e., $\forall q \in \mu(a) : q$ is correct. We assume that the owners of an account trust each other and, if any owner of an account is corrupted, the other owners are also considered corrupted (i.e., Byzantine) and thus lose any guarantees provided by the system.

2.3.3 Consensus objects

The owners of each correct account share an unbounded supply of *consensus objects* $\textsc{Consensus}[acc][1, 2, \dots]$. Each consensus object exports a single operation Propose(v), which satisfies the following three properties: (C-Liveness:) each invocation of Propose(v) by a correct client eventually returns a value; (C-Consistency:) no two correct clients return different values; and (C-Validity:) if a correct client returns value v, then some client invoked Propose(v).

The particular implementation of consensus can be chosen by the owners of the account and does not have to be trusted by other participants. As an extreme example, if an account is shared by two people, they could resolve a conflict via a phone call. Another option would be to use any consensus-based blockchain of choice.

2.3.4 Protocols and executions

A *protocol* equips every process p (client or replica) with an automaton A_p that, given an *input* (a received message or, if p is a client, an invocation of an operation), changes its state according to its transition function and produces an output (a message to send, a consensus invocation, or, if p is a client, a response to an operation). An *execution* of a protocol is a sequence of *events*, where each event is a received message, a sent message, an invocation, or a response. We assume that every invocation or response carries a unique identifier of the corresponding *operation instance* (we simply say operation).

2.3.5 Cryptographic Primitives

For simplicity, we assume the cryptographic primitives to be "oracles" implementing their ideal functionalities.

Digital signatures. In our algorithms we extensively use digital signatures to ensure that every participant can verify the authenticity of the received messages. We model digital signatures using two functions:

- Sign(m) – returns a signature for message m;
- Verify(m, sig, p) returns *true* iff *sig* was obtained with Sign(m) invoked by process p.

Threshold signatures. Additionally, we assume that $n-f$ valid signatures on the same message m can be efficiently aggregated via a *threshold signature scheme* [24, 124] or a *multi-signature scheme* [24, 117]. Namely, the following operations are available to all processes:

- CreateTS(m, S) – returns a threshold signature given a message m and a set S of valid digital signatures on message m issued by $n - f$ distinct replicas;
- VerifyTS(m, s) – returns *true* iff signature s was obtained by invoking CreateTS(m, S) for some set S.

Merkle trees. A Merkle tree (or a hash tree) is a binary tree in which every leaf node is labeled with a value (or a hash of a value), and every internal node is the hash of its two child nodes. One can use it to efficiently create a short commitment (namely, the root of the tree) to a set of values $M = \{m_1, \ldots, m_k\}$. Then, for any of the original values m_i, it is easy to prove that m_i belongs to M to anybody who knows the root. We model this primitive with the following functions available to all processes:

- MerkleTree(M) – returns a Merkle tree *merkleTree* for the set of values M. One can access the root of the tree using the notation *merkleTree.root*;
- GetItemProof(*merkleTree*, m) – returns a proof for item m iff *merkleTree* = MerkleTree(M) for some M s.t. $m \in M$;
- VerifyItemProof(*root*, *itemProof*, m) – returns *true* iff *itemProof* = GetItemProof(*merkleTree*, m) for some *merkleTree* with *merkleTree.root* = *root*.

In our protocols, Merkle trees could be replaced by more communication-efficient cryptographic primitives such as *set accumulators* [22] or *vector commitments* [41, 44]. However, they typically require more expensive computation and a trusted setup.

2.4 CryptoConcurrency Architecture

In a traditional, consensus-based, asset transfer system [112, 139], the participating processes agree on a (totally ordered) sequence of transactions, usually split into discrete blocks and applied to some initial state (often called *the genesis block*), as illustrated in Figure 2.1.

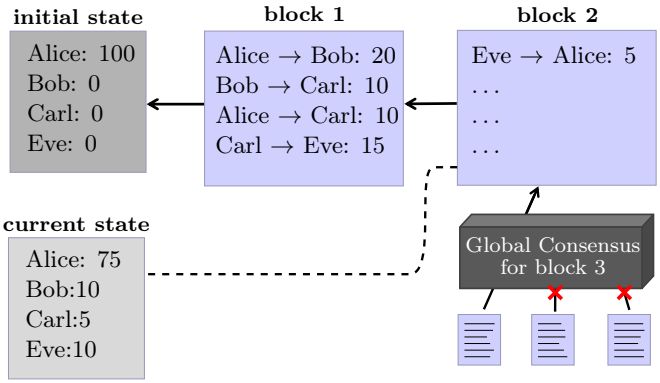

Figure 2.1: Total order asset transfer architecture

A crucial observation is that, as long as the final balance of each account is non-negative, the resulting state does not depend on the order in which the transactions are applied. This provides the core insight for the so-called *consensus-free* asset transfer systems [19, 48, 78, 79, 92, 126]. At a high level, such systems maintain an *unordered set* of committed transactions. In order to be added to the set, a new transaction must pass a special *Conflict Detector object*. The object maintains the invariant of non-negative balances by imposing a notion of pairwise conflicts on the transactions and preventing multiple conflicting transactions from being accepted (see Figure 2.2). Intuitively, two transactions are considered conflicting when they are trying to move the same assets. The Conflict Detector object operates in a way similar to Byzantine Consistent Broadcast [34, 48]. Namely, a quorum of replicas must acknowledge a transaction in order for it to pass the Conflict Detector and each replica acknowledges at most one of the conflicting transactions.

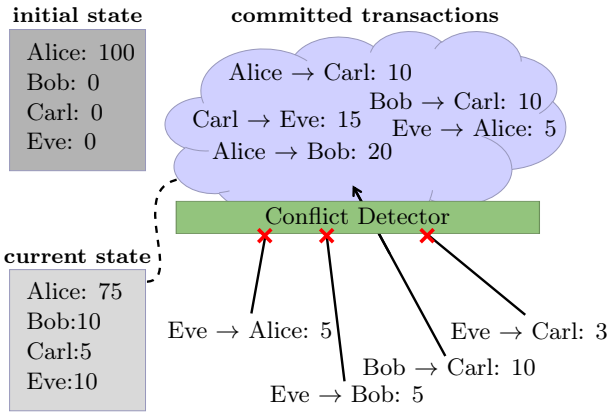

Figure 2.2: Consensus-free asset transfer architecture

As discussed in the introduction, the main downside of such systems is that they preclude any concurrent use of an account. Moreover, the existing solutions may actually punish even accidental attempts to issue several conflicting transactions concurrently by not letting *any* of them to pass the Conflict Detector and, hence, effectively blocking the entire account.

To mitigate this issue and enable new use-cases such as shared accounts or periodic subscription payments, we propose a hybrid approach: we replace the Conflict Detector with a more advanced *Recoverable Overspending Detector* abstraction and use external consensus objects to perform the recovery procedure in case an attempt to overspend is detected. We build an adaptive asset transfer system that goes through a consensus-free "fast path" whenever possible. The system supports shared accounts and avoids blocking the funds because of an accidental attempt of overspending on an account (see Figure 2.3).

In order to preserve the efficiency and robustness of consensus-free solutions, the consensus objects are used as rarely as possible. More precisely, our protocol only accesses consensus objects when the total volume of all ongoing transactions on the account exceeds the balance of the account. Our implementation

of the Recoverable Overspending Detector (see Section 2.6) inherits the key ideas from the Lattice Agreement protocol of [61] and Paxos [97, 98].

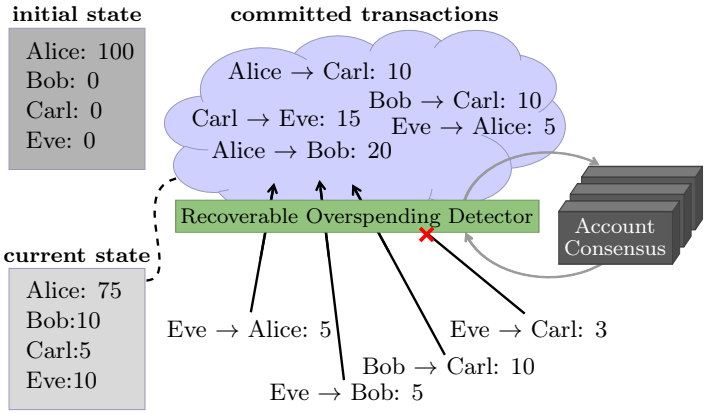

Figure 2.3: CryptoConcurrency architecture

Finally, we avoid the need of a central, universally trusted consensus mechanism by allowing the owners of each account to use their own consensus protocol of choice. To this end, after obtaining a consensus output, the owners send "notarization" requests to the replicas and the replicas will refuse to notarize diverging outputs for the same consensus instance.

2.5 Formal Problem Statement

Now we formally define the asset transfer abstraction that CryptoConcurrency implements.

Transactions. In asset transfer systems, clients move funds between accounts by issuing *transactions*. A *transaction* is a tuple $tx = \langle sender, recipient, amount, id, sig \rangle$. The *amount* value specifies the funds transferred from account *sender* to account *recipient*. In order to distinguish transactions with identical accounts and transferred amounts, each transaction is equipped with a special unique element called *id*. In practice, one can use a long (e.g., 128 bits) randomly generated string or a sequence number concatenated with the id of the client. Each transaction contains a digital signature *sig* of one of the owners of the account $tx.sender$ to confirm the transaction's authenticity. We denote the set of all possible well-formed transactions as $\mathcal{T}$. Ill-formed transactions (including transactions with invalid signatures) are ignored by the participants.

For every account $a \in \mathcal{A}$, there exists a *genesis transaction* $tx_{init,\ a} = \langle \bot, a, amount_a, 0, \bot \rangle$, which specifies the initial balance $amount_a$ of account a. All genesis transactions are publicly known and are considered to be well-formed by definition.

In addition, from the perspective of an account $a \in \mathcal{A}$, we distinguish two types of transactions: *debits* and *credits* (on a). A *debit transaction* (or simply a *debit*) is a transaction tx, for which $tx.sender = a$, and a *credit transaction* (or simply a *credit*) is a transaction tx, such that $tx.recipient = a$. In other words, debits "spend money" and credits "add money to the account".

Let us also define a helper function TotalValue that, given a set of transactions, returns the sum of funds they transfer: $\text{TotalValue}(txs) = \sum_{tx \in txs} tx.amount$. Let $\text{credits}(txs, acc)$ and $\text{debits}(txs, acc)$ denote the sets of credit and debit transactions on acc in txs, respectively (i.e., $\text{credits}(txs, acc) = \{tx \in txs \mid tx.recipient = acc\}$ and $\text{debits}(txs, acc) = \{tx \in txs \mid tx.sender = acc\}$). Now the *balance* of acc in txs is defined as: $\text{balance}(txs, acc) = \text{TotalValue}(\text{credits}(txs, acc)) - \text{TotalValue}(\text{debits}(txs, acc))$.

Interface. Clients interact with the asset transfer system using operations $\text{Transfer}(tx)$ and $\text{GetAccountTransactions}()$. The system also provides a function $\text{VerifyCommitCertificate}(tx, \sigma_{commit})$.

Operation $\text{Transfer}(tx)$, $tx \in \mathcal{T}$, is used by the clients to move assets as stipulated by the transaction tx. The operation may return one of the following responses:

- OK(σ_{commit}), indicating that the transfer has been completed successfully, σ_{commit} is a certificate proving this. σ_{commit} can be verified by any process using the VerifyCommitCertificate function;
- FAIL, indicating that the transfer failed due to insufficient balance.

Operation GetAccountTransactions() can be used to obtain the current set $\{\langle tx, \sigma_{commit}\rangle\}$ of debit and credit transactions tx applied to the client's account with their commit certificates σ_{commit}.

Committed and active transactions. A transaction tx is called *committed* iff there exists a certificate σ_{commit} such that VerifyCommitCertificate$(tx, \sigma_{commit}) = true$. For the purposes of this chapter, by the existence of a cryptographic certificate, we mean that some process or the adversary is capable of computing it with non-negligible probability using the available information in polynomial time. Let $C(t)$ denote the set of all such transactions at time t. Note that for all $t' > t$, we have $C(t) \subseteq C(t')$. We define commitTime(tx) as the moment of time t when tx gets committed, i.e., $t = \text{commitTime}(tx) \Leftrightarrow tx \in C(t)$ and $\forall t' < t : tx \notin C(t')$.

We assume that every transaction has a unique identifier and that the owners of a correct account will never invoke Transfer more than once with the same transaction tx. Hence, for a correct account acc, there is a one-to-one mapping between the debit transactions issued on acc and Transfer operations invoked by the owners of acc (including the ones that return FAIL).

From the perspective of the owners of a correct account acc, a debit transaction starts when the corresponding Transfer operation is invoked and ends when the operation terminates (with either OK(σ_{commit}) or FAIL). Hence, for a debit transaction tx, we define $\text{start}_{acc}(tx)$ and $\text{end}_{acc}(tx)$ as the moments in time when the corresponding operation is invoked and returns, respectively.

A debit transaction tx on a correct account is called *active at time* t iff $\text{start}_{acc}(tx) \leq t \leq \text{end}_{acc}(tx)$. Let $O(t, acc)$ denote the set of all active *debit* transactions on account acc at time t.

As for the credit transactions, the owners of acc have no insight into the execution of the corresponding operations (which could be performed by Byzantine clients). Instead, a credit transaction appears to happen instantly at the moment it is committed. Hence, for a credit transaction tx, we define $\text{start}_{acc}(tx) = \text{end}_{acc}(tx) = \text{commitTime}(tx)$.

Properties. Given an execution $\mathcal{E}$ and a correct account acc, we define $\mathcal{T}(\mathcal{E}, acc)$ as the set of all debit transactions and *committed* credit transactions on acc that appear in $\mathcal{E}$. The map $\rho_{\mathcal{E},acc}$ associates each debit transaction in $\mathcal{T}(\mathcal{E}, acc)$ with its response in $\mathcal{E}$ (if any). We say that a debit transaction $tx \in \mathcal{T}(\mathcal{E}, acc)$ is *successful* iff $\rho_{\mathcal{E},acc}(tx) = \text{OK}(\sigma)$ (for some σ).

We define a *real-time* partial order on transactions in $\mathcal{T}(\mathcal{E}, acc)$ as follows: we say that tx_1 *precedes* tx_2 in execution $\mathcal{E}$ from the point of view of account acc, and we write $tx_1 \prec_{\mathcal{E},acc} tx_2$ iff $\text{end}_{acc}(tx_1) < \text{start}_{acc}(tx_2)$ in $\mathcal{E}$.

Let H be a permutation (i.e., a totally ordered sequence) of transactions in $\mathcal{T}(\mathcal{E}, acc)$. Given tx, a debit transaction on acc in $\mathcal{E}$, let $S(H, tx)$ denote the set of credit and successful debit transactions in the prefix of H up to, but not including, tx. We say that permutation H is *legal* if and only if, for every debit transaction $tx \in \mathcal{T}(\mathcal{E}, acc)$, $\rho_{\mathcal{E},acc}(tx) = \text{OK}(\sigma) \Leftrightarrow tx.amount \leq \text{balance}(S(H, tx), acc)$.

We say that H is *consistent with* $\prec_{\mathcal{E},acc}$ iff for all $tx_1, tx_2 \in \mathcal{T}(\mathcal{E}, acc)$, $tx_1 \prec_{\mathcal{E},acc} tx_2$ implies that tx_1 precedes tx_2 in H.

Now we are ready to formally state the properties that every execution $\mathcal{E}$ of our *asset transfer implementation* must satisfy. First, no account (be it correct or Byzantine) can exhibit a negative balance:

Transfer Safety: At any time t, for all $acc \in \mathcal{A}$: $\text{balance}(C(t), acc) \geq 0$.

Furthermore, for every correct account acc, from the point of view of the owners of the account, the outputs of the Transfer operations are as if they were executed sequentially, one at a time, with no concurrency and the certificates returned by Transfer are valid. More formally, the following properties hold for each correct account acc and each protocol execution $\mathcal{E}$:

Transfer Consistency: There exists a legal permutation of transactions in $\mathcal{T}(\mathcal{E}, acc)$ that is consistent with $\prec_{\mathcal{E},acc}$.

Transfer Validity: If Transfer(tx) on acc returns OK(σ_{commit}), then VerifyCommitCertificate(tx, σ_{commit}) = *true*.

The second operation, GetAccountTransactions, must return the set of committed transactions related to the account:

Account Transactions Completeness: GetAccountTransactions() invoked by an owner of a correct account acc at time t_0 returns a set $\{\langle tx_i, \sigma_i \rangle\}_{i=1}^{l}$ such that $\forall i$: VerifyCommitCertificate(tx_i, σ_i) = *true* and debits($C(t_0), acc$) $\cup$ credits($C(t_0), acc$) $\subseteq \{tx_i\}_{i=1}^{l}$.

An asset transfer system should also satisfy the following liveness property:

Transfer Liveness: Every operation invoked by a correct client eventually returns.

Finally, CryptoConcurrency satisfies one more important property that we consider as one of the key contributions of this chapter. Intuitively, if the owners of a correct account do not try to overspend, eventually the system will stabilize from their previous overspending attempts (if any), and the clients will not need to invoke consensus from that point on. Let us now define this property formally.

We say that there is an *overspending attempt at time t* iff TotalValue($O(t, acc) \setminus C(t)$) > balance($C(t), acc$).

Transfer Concurrency: Let acc be a correct account. If there is no overspending attempt at any time $t > t_0$, for some t_0, then there exists a time t_1 such that the owners of acc do not invoke consensus objects after t_1.[1]

In terms of algorithmic complexity, this chapter is focused on the latency exhibited by an asset-transfer implementation in the absence of overspending attempts, i.e., when consensus objects are not involved. We measure the latency in *round-trip times* (RTTs). Informally, RTT is the time it takes for a given process to send a *request* message to another process and receive a *response* message. [2] For an algorithm satisfying the Transfer Concurrency property, we say that it exhibits *k-overspending-free latency* $f(n, k)$ if, after the time t_1 (defined in Transfer Concurrency), any transfer operation that runs in the absence of overspending concurrently with at most $k - 1$ other transfer operations on the same account in a system with n replicas completes in at most $f(n, k)$ RTTs.

Given the above, the main theorem of this chapter is as follows.

Theorem 2.1. *There exists a deterministic asynchronous protocol that implements an asset transfer system (as formally defined in this section), satisfies the Transfer Concurrency property and exhibits k-overspending-free latency of* $k + 4$ *RTTs.*

2.6 Closable Overspending Detector

We implement the Recoverable Overspending Detector layer illustrated in Figure 2.3 in Section 2.4 as a collection of slightly simpler objects, which we call *Closable Overspending Detector* (or *COD* for short). COD objects are account-specific. At each moment in time, there is at most one COD object per account that is capable of accepting client transactions. The mission of a single COD object is to ensure operation under normal conditions when there are no overspending attempts. In this case, COD will accept every transaction that a client submits to it using the Submit operation.

However, because of concurrency, even the owners of a correct account might accidentally try to overspend. In this case, some of the client requests submitted to COD may fail. Clients can then use the Close operation that deactivates this COD instance and provides a snapshot of its final state. This allows the account owners to gracefully recover from overspending by instantiating a new instance of COD from this snapshot. To this end, each account acc is provided with a list of COD objects COD$[acc][1, 2, \ldots]$. Object COD$[acc][e]$ is said to be associated with *epoch number* e. The procedure of migrating from one COD object to another, i.e., from one *epoch* to another, is called *recovery* and will be described in detail in Section 2.8.1.

[1] Note that, if the Transfer Consistency property holds, all Transfer operations invoked after t_1 must return OK(σ_{commit}) and cannot return FAIL.

[2] A more precise definition of time complexity of an asynchronous algorithm can be found in [39].

2.6.1 COD Protocol Overview

We define the interface and the properties of COD formally and provide pseudocode in Section 2.10. However, learning about this concept alongside a high-level overview of the algorithm, as presented in the rest of this section, may be more accessible. In order to keep the explanation simple and highlight the main ideas, we omit some minor implementation details.

Initial state. As described before, each account is associated with a sequence of COD objects, each serving for one *epoch* – a period of time without overspending attempts. However, if an overspending attempt is detected, a COD object is closed, and, after a procedure that we call *recovery*, a new COD object is initialized. Thus, a COD object needs to be initialized from some initial state, namely: *initDebits* – the set of debits accepted in prior epochs, $initCredits^{\sigma}$ – a set of credits with commit certificates sufficient to cover *initDebits*, and *restrictedDebits* – the set of debits canceled in prior epochs (the COD object must not accept the transactions from *restrictedDebits*).

Submit operation. The main operation of COD is Submit($debits, credits^{\sigma}$), through which clients inform replicas of new incoming transactions (credits) and request approval for new outgoing transactions (debits). Credits must be accompanied by valid commit certificates. Hence, $credits^{\sigma}$ is a set of pairs $\langle tx, \sigma_{commit} \rangle$ such that $tx.recipient = acc$ and VerifyCommitCertificate$(tx, \sigma) = true$.

As a convention, throughout the rest of the chapter, we use variable names with a superscript "σ" (e.g., txs^{σ}, $credits^{\sigma}$, $debits^{\sigma}$, etc.) to denote sets of transactions paired with some kind of cryptographic certificates (e.g., $\{\langle tx_1, \sigma_1 \rangle, \ldots, \langle tx_n, \sigma_n \rangle\}$). We also define an auxiliary function Txs(txs^{σ}) that, given a set of pairs $txs^{\sigma} = \{\langle tx_1, \sigma_1 \rangle, \ldots \langle tx_n, \sigma_n \rangle\}$, returns only the transactions $\{tx_1, \ldots, tx_n\}$, without the certificates.

In the optimistic scenario, when there are no overspending attempts, Submit returns OK($debits^{\sigma}, outCredits^{\sigma}$). Here, $debits^{\sigma}$ contains the same set of transactions as in the input to Submit, augmented with certificates that we will call *accept certificates*, confirming that the transactions passed through the Closable Overspending Detector object. The COD implementation exposes a boolean function VerifyCODCert(tx, σ) that can be used to verify the accept certificates. We say that a transaction tx is *accepted by COD*$[acc][e]$ iff $tx \in$ COD$[acc][e].initDebits$ or there exists σ such that COD$[acc][e]$.VerifyCODCert$(tx, \sigma) = true$.

The second value returned by a successful invocation of Submit, $outCredits^{\sigma}$, is, intuitively, the set of credits used to "cover" the debits. For correct accounts, COD maintains the property that, at any time t, the total value of all debits accepted by COD does not exceed the total value of all credits returned from successful invocations of Submit.[3]

The "Prepare" phase. To detect overspending attempts, a correct replica r maintains a set $credits_r^{\sigma}$ of all credits it has seen so far (with the corresponding commit certificates) and a set $debits_r^{\sigma}$ of debit transactions it acknowledged (with client signatures), preserving the invariant that TotalValue(Txs($credits_r^{\sigma}$)) $\geq$ TotalValue(Txs($debits_r^{\sigma}$)). To process a client request, it adds all the committed credits the client attached to the message to the local set $credits_r^{\sigma}$ (given they come with valid commit certificates) and then adds the received debits to $debits_r^{\sigma}$ if possible without violating the invariant. The replica then responds to the client with both sets and a signature on the debits set.

Then, on the client side, it is tempting to wait for a quorum of responses, such that each will contain client debit transactions and assume that this means that the set of transactions client sent to replicas does not lead to overspending. Indeed, it would mean that at least $\frac{n}{3}$ correct replicas added all input debits to their local sets. However, this approach is not sufficient to prevent potential overspending. Consider the example illustrated in Figure 2.4. In this example, each of the 3 correct replicas (1,2, and 4) acknowledged 2 transactions each and thus did not detect overspending. The third replica is Byzantine and it acknowledged all 3 transactions. In the end, all three transactions would manage to pass the overspending detector even though the account they share contains funds only for 2 of them.

To deal with such situations, we follow a similar approach to the one proposed to solve Generalized Lattice Agreement [61]: we retry requests to replicas until we receive identical sets of debits in the

[3]For Byzantine clients, we cannot formally use the language of "returned values". Instead, for Byzantine accounts, COD guarantees that the total value of accepted debits does not exceed the total value of *all* committed credits for the account, ensuring non-overspending.

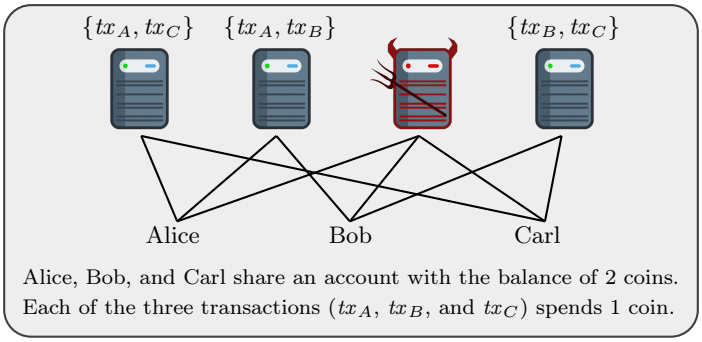

Figure 2.4: An example where the naive 1 RTT algorithm fails.

responses, updating the inputs with the new transactions we learn with every attempt. In the above example, Alice, after receiving $\{tx_A, tx_C\}$ from replica 1 and $\{tx_A, tx_B\}$ from replica 2, would retry its request as these two responses are different even though both contain tx_A. However, unlike in Generalized Lattice Agreement, where the assumption is that any combination of inputs is "mergeable" (i.e., commutative), in our case, due to the non-overspending invariant maintained by the replicas, the object may reach a state where convergence would not be reached regardless of how many retries the client performs. We will explain how to deal with such situations later in this section. For now, let us consider the "good" scenario when there are no overspending attempts.

For all requests that successfully passed this phase, we can guarantee the *comparability* property: suppose that client A obtains a quorum of signatures for a set of transaction S_A and client B – for a set S_B. Then we can claim that either $S_A \subseteq S_B$ or vice versa. Indeed, it is sufficient to consider the quorum intersection property [107]: there must be a correct replica that signed both sets and the one it signed later must be a superset of the one it signed earlier. We can also guarantee that for all transactions in both sets, there are enough committed credits to cover all of them. Indeed, consider the largest of the two sets. It was acknowledged by a quorum of replicas that would only acknowledge it if they saw enough credits.

We call this process of repeatedly trying to get a quorum of replicas to converge on the same set of debits the *Prepare phase*. After a client successfully passes the Prepare phase in the course of executing a Submit operation, the client obtains a set of debits *preparedDebits* with signatures from a quorum of replicas confirming that they acknowledged this set. We use a *threshold signature scheme* (as defined in Section 2.3.5) in order to compress these signatures into one small signature $\sigma_{prepare}$.

The "Accept" phase. The purpose of the second phase of the Submit operation, called the *Accept phase*, is to ensure *recoverability*, i.e., to make it possible to transfer state from one COD object to the next without reverting transactions that could have been accepted. The phase consists of just one round-trip: the client simply sends *preparedDebits* along with the threshold signature $\sigma_{prepare}$ to the replicas, the replicas check the validity of $\sigma_{prepare}$ and, *unless they have previously received a request to close this COD instance*, acknowledge the client's request with a signature on the *Merkle tree root* (as defined in Section 2.3.5) of the set *preparedDebits*. The client can then extract individual Merkle tree proofs for each of the transactions, thus obtaining a short accept certificate for each individual transaction.

Detecting overspending. The client may not be able to terminate its request on the "good" path for one of two reasons:

1. In the Prepare phase, the client does not have enough committed credits to cover the union of sets of debits returned by the replicas. This means that there is an overspending attempt;

2. In either phase, a replica refuses to process the client's request because it has already acknowledged a request to close this COD instance. This means that some other owner of the account observed an overspending attempt and started migrating the state from this instance to the next.

In either of these two cases, the client returns FAIL from the Submit operation, indicating a potential overspending attempt and that a (consensus-based) recovery is necessary.

Closing an instance. The Close operation is designed to deactivate the COD object and to collect a snapshot of its state to facilitate the recovery. During this operation, the client solicits from a quorum of replicas the sets of transactions they accepted and then asks the replicas to sign the accumulated joint state in order to obtain a short proof (a single threshold signature) of validity of the resulting snapshot. Accessing a quorum guarantees that the client gathers all the debits that have been previously returned from the Submit operations. As a result, accepted transactions are never lost.

Upon careful examination, one can notice that the interaction between the Accept phase of the Submit operation and the first message of the Close operation is similar to that of "propose" messages and "prepare" messages with a larger ballot number in Paxos [98]. Intuitively, it guarantees that, if there is a Submit concurrent with a Close, either the client executing Submit will "see" the Close operation and return FAIL or the client executing the Close operation will "see" that the debits were accepted by some replicas.

Achieving liveness. Special care is necessary to ensure liveness of all clients in such a protocol, especially in the Prepare phase. First, we only wait for convergence on debits in the replicas' responses and not on credits, as otherwise, it would be possible to prevent progress on a correct account by sending lots of small credits to it. Second, it is not hard to see that a certain unlucky client may never reach convergence due to a constant inflow of new debits. In order to avoid such situations, a replica that already accepted a set of debits that includes all the debits submitted by a client will notify that client, and it will be able to move on directly to the Accept phase.

2.6.2 COD performance

We are mostly interested in the latency of the Submit operation as the Close operation is only used in case of an overspending attempt, which is assumed to happen rarely. The protocol for the Submit operation consists of two phases: Prepare and Accept, where the latter always consists of just one rount-trip while the former may involve multiple retries, until the client obtains a quorum of identical replies. However, the main strength of this implementation is that it is adaptive. Indeed, in absence of other concurrent requests, the client will be able to finish the Prepare phase in just one round-trip. Otherwise, it may take up to k round-trips in presence of $k-1$ other concurrent requests.

Put differently, we pay one extra round-trip compared to purely asynchronous solutions such as [19, 48] in order to ensure recoverability (the Accept phase), but the overspending detection part comes "for free". The main difference is that CryptoConcurrency retries where others would give up, until it reaches a state where further retries would be pointless.

2.7 Append-Only Storage

In this section, we present an abstraction called *Append-Only Storage*, which allows us to implement two distinct algorithm building blocks. The first one, *Global Storage*, represents a layer in the architecture of CryptoConcurrency that stores all committed transactions (as illustrated in Figure 2.3 in Section 2.4). The second one, Account Storage, is associated with each account, and its primary purpose is to facilitate communication between its owners and ensure liveness of its operations.

Append-Only Storage can be seen as a distributed implementation of an indexed collection of sets: with each *key* k, the abstraction associates an unordered set of *values* vs. Furthermore, Append-Only Storage is capable of (i) verifying that a given value is allowed to be added to a specified key and (ii) providing a proof of the fact that the values appended or read by a client for a specific key k are stored persistently (i.e., any later read operation with key k will return a set that includes these values).

To add a value v to a set of values stored for a key k, a client calls AppendKey(k, v, σ_v), where σ_v is a validity certificate for value v and key k. To read the values associated with a key k, a client invokes the ReadKey(k) operation.

We provide a formal definition of Append-Only Storage and discuss the protocol implementing it in Section 2.11. For simplicity, we use a highly consistent storage system that requires only one round-trip for the AppendKey operation, but two round-trips for the ReadKey operation: one to fetch the values and one for the "write-back" phase [17] to ensure that any subsequent read will see at least as many elements in the set.

2.7.1 Global and Account Storage

We use Append-Only Storage as a generalized implementation for both *Global Storage* and *Account Storage*. Let us explain how we use each of these objects in the protocol.

Global Storage. In CryptoConcurrency, all clients have access to one common Append-Only Storage instance called *Global Storage*. After a transaction is accepted by a COD object or the recovery procedure (discussed in more detail below), it is written to *Global Storage* to make it publicly available to all clients. In CryptoConcurrency, the moment a transaction is written to Global Storage, it becomes committed and the persistence certificate plays the role of a commit certificate for the transaction. One can view the Global Storage abstraction as the "final" public transaction ledger. However, unlike in consensus-based cryptocurrencies, this ledger is not a sequence, but just a set, and the transactions in it can be applied in any order. In the pseudocode, we denote this instance by GLOBALSTORAGE.

Account Storage. Every account *acc* is equipped with an instance of Append-Only Storage called *Account Storage*. Inside this instance, clients that share account *acc* store information about started debit transactions on *acc* and initial states for COD objects. This allows us to ensure progress of all operations on the account: i.e., any operation invoked by an owner of *acc* terminates (assuming *acc* is correct). More specifically, we use the technique known as *helping*, common to concurrent algorithms [9, 77, 81]. Communicating via storage allows clients to temporarily go offline (lose connection) and still preserve all system guarantees, which would be hard to achieve with a broadcast-like algorithm. In the code, we denote an instance of an Account Storage for an account *acc* as ACCOUNTSTORAGE[*acc*].

Similarly to the consensus objects, the implementation of Account Storage can be account-specific and only the owners of the account need to trust it.

2.8 CryptoConcurrency: Algorithm

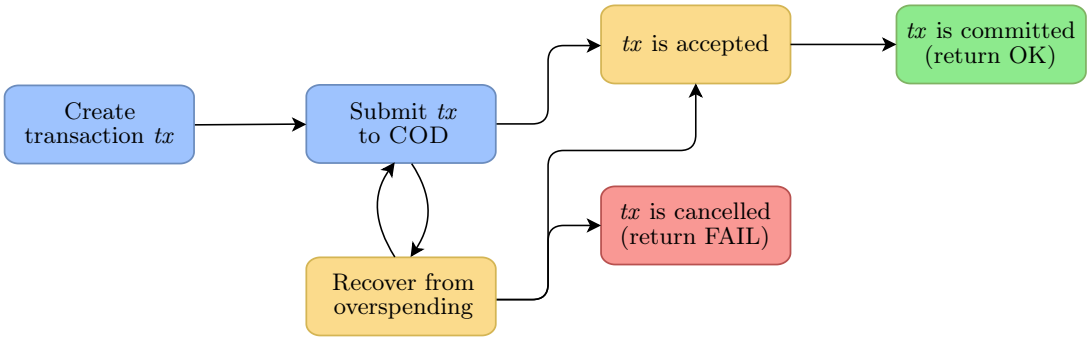

Figure 2.5: Transaction lifecycle in CryptoConcurrency

In this section, we demonstrate how to combine Closable Overspending Detector, Account Storage, Global Storage, and Consensus objects in order to obtain a protocol implementing an asset transfer system with the Transfer Concurrency property as defined in Section 2.5, thus providing a constructive proof for Theorem 2.1.

2.8.1 Composing COD and Consensus instances

In CryptoConcurrency, COD serves as a fundamental building block: it allows the owners of an account *acc* to issue concurrent transactions. As long as there are enough committed credit transactions to cover all debit transactions submitted to COD, all owners of a correct account will be able to confirm their transactions and get matching certificates, i.e., the COD object will accept them.

However, even correct clients may accidentally try to overspend. In this case, a correct client might return FAIL from a Submit operation. Intuitively, when this happens, we consider the internal state of the COD instance being "broken" and we need to "recover" from this.

Let us overview the recovery procedure performed after an overspending attempt was detected in COD[*acc*][*e*]. First, the client invokes the Close operation on COD[*acc*][*e*], which will return a snapshot with the following data:

1. a set of *selected debits* that includes (but is not limited to) all debit transactions accepted by COD[*acc*][*e*];
2. a set of committed credit transactions sufficient to cover the selected debits;
3. a set of *cancelled debits*, for which there are not enough known credits.

If multiple clients invoke Close, the snapshots they receive might vary due to the asynchronous nature of COD. However, *every* valid snapshot will include *all* debit transactions that are or ever will be accepted by COD[*acc*][*e*]. To this end, the Close operation "invalidates" COD[*acc*][*e*], so that no transaction can be accepted after a snapshot was made (in case of concurrency, either the transaction will make it to the snapshot or it will not be accepted).

In order to initialize COD[*acc*][$e+1$], the clients must agree on its initial state. Hence, they will propose the snapshots they received from the Close operation invoked on COD[*acc*][*e*] to Consensus[*acc*][$e+1$]. There will be exactly one snapshot selected by the consensus instance, from which the initial state for COD[*acc*][$e+1$] will be derived. We would like to emphasize that CryptoConcurrency preserves both safety and liveness regardless of which of the (potentially multiple) valid snapshots is selected by the consensus object as long as the owners of the account agree on it (which is guaranteed by the C-Consistency property of consensus).

We do not bind clients to any specific consensus protocol: every instance of consensus can be implemented differently. To preserve safety, the consensus output must be *notarized* (i.e., signed) by a quorum of replicas that run CryptoConcurrency, and no correct replica will sign two different consensus outputs. Since the owners of a correct account will never try to get multiple different outputs notarized for the same consensus instance, they will always be able to get the notarization. The consensus output signed by a quorum can be used to initialize COD[*acc*][$e+1$]. This concludes the recovery procedure.

2.8.2 Transfer algorithm

Overview. At a very high level, when the Transfer(*tx*) operation is invoked, *tx* is submitted to the latest COD instance. If there are sufficient credit transactions on the account to cover *tx*, it is accepted by the COD instance and registered in the Global Storage. Otherwise, there is a risk of overspending, and the account's state is recovered as described in Section 2.8.1. A transaction is considered *failed* if it ends up in the set of cancelled debits in the snapshot selected by the consensus object. Similarly, a transaction can be accepted by the recovery procedure if it ends up in the set of selected debits. Figure 2.5 depicts the lifecycle of a transaction in our protocol.

Preparation. The algorithm proceeds in consecutive epochs for each account. For a correct account *acc*, epoch *e* corresponds to the period of time when COD[*acc*][*e*] is active (i.e., is initialized but is not yet closed). Let us consider a correct client *p*, one of the owners of an account *acc*. When *p* invokes Transfer(*tx*), it first fetches the current epoch number *e* and the initial state of COD[*acc*][*e*] from the AccountStorage[*acc*]. The client also accesses the Global Storage to fetch all newly committed credit transactions for account *acc*. It will later submit these credit transactions to the COD object along with *tx*. Lastly, *p* writes *tx* to AccountStorage[*acc*]. This way, other owners of *acc* will be able to help commit *tx* when they read it from the Account Storage. This step is crucial to avoid starvation of slow clients.

Main loop. After the preparation, the client enters a loop. In each iteration (corresponding to one *epoch e*), the client reads the pending debits (all debit transactions on account *acc* that are not yet committed) from the Account Storage. After this, *p* invokes the Submit operation on COD[*acc*][*e*] with the pending debits it just read from the Account Storage and the committed credits it read during the preparation phase from the Global Storage. If the invocation returns OK(...), then the only thing left to do is to commit *tx* by writing it to the Global Storage (as a result, the recipients of the transaction can learn about it by reading the Global Storage). Finally, *p* returns OK from the Transfer operation

with the certificate of a successful write to the Global Storage acting as the commit certificate σ_{commit} for transaction tx.

If, however, COD$[acc][e]$.Submit returns FAIL, it indicates that either the volume of all known credits was insufficient to cover all existing debits at some point of time, or some other client invoked the Close operation on this COD object. Either way, the client then goes through the recovery procedure as described in Section 2.8.1 in order to obtain the initial state for COD$[acc][e+1]$.

Finally, the client checks whether, during the Recovery procedure, tx ended up in the set of selected or cancelled debits (see Section 2.8.1). In the former case, the client commits the transaction by writing it to the Global Storage and returns OK. In the latter case, the client simply returns FAIL from the Transfer operation. However, it might also happen that tx is neither accepted nor canceled (e.g., if, due to concurrency, the initial state for COD$[acc][e+1]$ was selected before any process other than p learned about tx). In this case, p increments its local epoch number e and proceeds to the next iteration of the loop, proposing tx to the next COD instance.

This loop eventually terminates due to the helping mechanism: once tx is stored in the Account Storage in the preparation phase, every other owner of acc executing Transfer submits tx to COD along with their own transactions.

2.8.3 CryptoConcurrency pseudocode

The pseudocode for CryptoConcurrency is provided in Algorithms 2.1 to 2.3 and algorithm 4. We assume that each block of code (a function, an operation, a procedure or a callback) is generally executed sequentially to completion. However, a block may contain a **wait for** operator, which interrupts the execution until the wait condition is satisfied. Some events, e.g., receiving a message, might trigger callbacks (marked with **upon** keyword). They are not executed immediately but are first placed in an event queue, waiting for their turn (we assume a fair scheduler). We denote an assignment of an expression *expr* to a variable *var* as $var := expr$.

In the code, we assume that each message sent by a client carries a distinct sequence number. When a replica replies to a client, it also implicitly includes the same sequence number in its response. This allows clients to match replies with the corresponding request messages and ignore outdated replies.

A message from process p is considered *valid* by process q if there exists a possible execution of the protocol in which p is correct and it sends this message to q. In most cases, this boils down to the correct number and order of message attachments as well as all the attached signatures and certificates being valid. In our protocols, we implicitly assume that invalid messages are simply ignored by correct processes. Hence, for the adversary, sending an invalid message is equivalent to not sending anything at all.

Recall that the m-th consensus instance associated with account acc is denoted by Consensus$[acc][m]$.

Clients and replicas in CryptoConcurrency. The client's protocol is described in Algorithms 2.1 and 2.2 and the replica's protocol is described in Algorithm 2.3. The algorithm uses instances of COD, and instances of Append-Only Storage: Global Storage (one per system, shared by all clients), and Account Storage (one per account, shared by the owners of the account). We describe objects initialization in Algorithm 4. The implementations of COD and Append-Only Storage are delegated to Sections 2.10 and 2.11, respectively.

The algorithm proceeds in consecutive epochs for each account. Let us consider a correct client p, one of the owners of a correct account acc.

When Transfer(tx) is invoked, client p first fetches the current epoch number e, the initial state *CODState* of COD$[acc][e]$ and a certificate σ for *CODState* from AccountStorage$[acc]$ (line 5). Then it reads committed transactions from Global Storage at line 7. After this, the client forms a set of credits (line 8) that are later submitted to the COD and writes tx to Account Storage (line 10): this way, other owners of acc will be able to help to commit tx.

Accessing COD After this, the client enters a while loop. In each iteration of the loop (corresponding to one *epoch* e), the client reads pending debits on acc from Account Storage (lines 13 and 15). Then, it sends the epoch number e, the initial state *CODState*, and the matching certificate σ to the replicas (line 17). This way replicas that fall behind can initiate an up-to-date instance of COD object for account

acc before receiving messages associated with it. After this, client p invokes the Submit operation on the corresponding COD object. If the invocation returns OK(...), then the only thing left to do is to write $\langle tx, \sigma \rangle$ to the Global Storage at line 23 (as a result, the recipients of the transaction can learn about it by performing GetAccountTransactions() on their side). Finally, p returns OK with the commit certificate from the Transfer operation.

Otherwise, if p returns FAIL from Submit, then either all known credits were not enough to cover all existing debits at some point of time, or some other client invoked Close operation on this COD object. Either way the client then goes through the Recovery procedure to get the state for the next $COD[acc][e+1]$ (line 26).

Recovery and consensus. In the Recovery procedure, the client first makes sure that the current instance COD[*acc*][*e*] is closed by invoking Close operation on it (line 44). Then it uses consensus object CONSENSUS[*acc*][*e*+1] (line 47) in order to initialize the next instance of COD. We allow the owners of each account to use local, distinct consensus objects that can be handled outside of the CryptoConcurrency system. However, due to this, the outcome of consensus should be signed by a quorum of replicas (lines 50-52 and 70-77), so that Byzantine clients are not able to break the system and use a diverging result.

The resulting initial state for the COD object in epoch $e+1$ contains *selectedDebits*, a set of transactions that were decided to be used as initial debits in the new transactions. It is guaranteed that all previously committed debits and all debits accepted by the COD[*acc*][*e*] are in this set, though it may also contain some extra debits. For convenience, the Recovery procedure also returns $selectedDebits^{\sigma}$ (signed version of *selectedDebits*) that are ready to be written to Global Storage. For every $\langle tx, \sigma \rangle \in selectedDebits^{\sigma}$: VerifyRecoveryCert$(tx, \sigma) = true$, where VerifyRecoveryCert is a publicly known function.

Finally, the client checks if tx is in the set *CODState.selectedDebits* (line 27) or *CODState.cancelledDebits* (line 31). In the former case, the client commits the transaction by appending it to the Global Storage (line 29) and returns OK. In the latter case, it simply returns FAIL.

However, it might happen that tx is in neither of these sets. In this case, p proceeds to the next iteration of the while loop corresponding to the next epoch number $e+1$ (line 33).

Algorithm 2.1 CryptoConcurrency (for client p, account acc)

```
1: type Σ := {0,1}*   // set of all possible cryptographic certificates
2: type T_Σ := Pair⟨T, Σ⟩      // set of all possible pairs of form ⟨tx, σ⟩, where tx ∈ T and σ ∈ Σ

3: operation Transfer(tx) returns OK(Σ) or FAIL
4:     // Read the latest up-to-date state of the account
5:     ⟨e, CODState, σ_state⟩ := ReadLatestCODState()
6:     // Read all committed transactions (with their commit certificates) related to this account
7:     committedTxs^σ := GetAccountTransactions()
8:     newCredits^σ := {⟨tx', σ_tx'⟩ | ⟨tx', σ_tx'⟩ ∈ committedTxs^σ, tx'.recipient = acc} \ CODState.initCredits^σ
9:     // Make sure that all other owners of this account will eventually see this transaction to prevent starvation
10:    AccountStorage[acc].AppendKey("debits", tx, ⊥)
11:    while true do
12:        // Read debits from the Account Storage, no need for certificates
13:        debits := Txs(AccountStorage[acc].ReadKey("debits"))
14:        // The client must help to commit other pending transactions in order to avoid starvation
15:        pendingDebits := debits \ CODState.cancelledDebits
16:        // Help all replicas to catch up with the current epoch and initialize COD[acc][e]
17:        send ⟨InitCOD, e, CODState, σ_state⟩ to all replicas
18:        // Try to commit the pending transactions (including tx) in this epoch
19:        CODResult := COD[acc][e].Submit(pendingDebits, newCredits^σ)
20:        if CODResult is OK(acceptedDebits^σ, outCredits^σ) then
21:            // Extract the certificate for tx from the Submit response
22:            let σ_COD be a certificate such that ⟨tx, σ_COD⟩ ∈ acceptedDebits^σ
23:            σ_commit := GlobalStorage[acc].AppendKey("txs", tx, ⟨acc, e, σ_COD⟩)
24:            return OK(σ_commit)
25:        // COD[acc][e].Submit has failed
26:        ⟨CODState, σ_state, selectedDebits^σ⟩ := Recovery(e, pendingDebits)
27:        if tx ∈ CODState.selectedDebits then
28:            let σ_recovery be a certificate such that ⟨tx, σ_recovery⟩ ∈ selectedDebits^σ
29:            σ_commit := GlobalStorage[acc].AppendKey("txs", tx, σ_recovery)
30:            return OK(σ_commit)
31:        if tx ∈ CODState.cancelledDebits then
32:            return FAIL      // The transaction is cancelled due to insufficient balance
33:        e += 1
34:        // CODState and σ_state from the recovery are used for the next iteration of the loop

35: operation GetAccountTransactions() returns Set⟨T_Σ⟩
36:    committedTxs^σ := GlobalStorage[acc].ReadKey("txs")
37:    return {⟨tx, σ_commit⟩ ∈ committedTxs^σ | tx.sender = acc or tx.recipient = acc}

38: public function VerifyCommitCertificate(tx, σ_commit) returns Boolean
39:    return GlobalStorage.VerifyStoredCert("txs", tx, σ_commit)

40: function ReadLatestCODState() returns ⟨EpochNum, CloseState, Σ⟩
41:    return ⟨e, CODState, σ⟩ such that ⟨⟨e, CODState, σ⟩, ⊥⟩ ∈ AccountStorage[acc].ReadKey("state")
   and e is maximum
```

Algorithm 2.2 CryptoConcurrency: Recovery (code for client p, account acc)

42: **function** Recovery(e, *pendingDebits*) **returns** Tuple$\langle$CloseState, Σ, Set$\langle\mathcal{T}_\Sigma\rangle\rangle$
43: // Close the current instance of COD and get the closing state and a certificate confirming that this state is valid
44: $\langle$*CODState*, *closedStateCert*$\rangle$:= COD[*acc*][*e*].Close(*pendingDebits*)
45: $\langle$*allCredits*$^\sigma$, *selectedDebits*, *cancelledDebits*$\rangle$:= *CODState*
46: // The consensus mechanism resolves the overspending attempts that led to the recovery.
47: $\langle$*nextCODState*, *nextCODStateCert*$\rangle$:= Consensus[*acc*][*e* + 1].Propose($\langle$*CODState*, *closedStateCert*$\rangle$)
48: // Since consensus is only trusted by the owners of the account, the client needs to commit to the consensus output.
49: // This prevents malicious clients from creating multiple different initial states for the same COD.
50: **send** $\langle$CommitInitState, $e+1$, *nextCODState*, *nextCODStateCert*$\rangle$ **to** all replicas
51: **wait for** valid $\langle$CommitInitStateResp, $sigState_i$, $sigTxs_i\rangle$ replies from a quorum Q
52: σ_{state} := CreateTS($\langle$CommitInitStateResp, *nextCODState*$\rangle$, $\{sigState_i\}_{i\in Q}$)
53: // Let other owners of the account know about the new epoch and its initial state.
54: AccountStorage.AppendKey("state", $\langle e+1$, *nextCODState*, $\sigma_{state}\rangle$, $\bot$)
55: // Create confirm certificates for the selected debits.
56: *merkleTree* := MerkleTree(*nextCODState.selectedDebits*)
57: σ_{MT} := CreateTS($\langle$ConfirmInRecovery, *e*, *merkleTree.root*$\rangle$, $\{sigTxs_i\}_{i\in Q}$)
58: *selectedDebits*$^\sigma$ = $\{\langle tx, \langle$*merkleTree.root*, GetItemProof(*merkleTree*, *tx*), σ_{MT}, *e*$\rangle\rangle$ | *tx* $\in$ *nextCODState.selectedDebits*$\}$
59: **return** $\langle$*nextCODState*, σ_{state}, *selectedDebits*$^\sigma\rangle$

60: **public function** VerifyRecoveryCert(*tx*, $\sigma_{recovery}$) **returns** Boolean
61: $\langle$*root*, *itemProof*, σ_{MT}, *e*$\rangle$:= $\sigma_{recovery}$
62: **return** VerifyItemProof(*root*, *itemProof*, *tx*) **and** VerifyTS($\langle$ConfirmInRecovery, *e*, *root*$\rangle$, σ_{MT})

Algorithm 2.3 CryptoConcurrency (code for replica r)

63: **State:**
64: *signedStates* – mapping from an account and an epoch number to the hash of a signed state,
initially: $\forall a \in \mathcal{A}, s \geq 0$: *signedStates*[*a*][*s*] = $\bot$

65: **upon receive** $\langle$InitCOD, *e*, *CODState*, $\sigma\rangle$ **from** owner q of account *acc*
66: **if** COD[*acc*][*e*] is already initialized **then return**
67: **if** not VerifyTS($\langle$CommitInitStateResp, *CODState*$\rangle$, σ_{state}) **then return**
68: $\langle$*allCredits*$^\sigma$, *selectedDebits*, *cancelledDebits*$\rangle$:= *CODState*
69: **initialize** COD[*acc*][*e*] **with**
initDebits := *selectedDebits*,
initCredits$^\sigma$:= *allCredits*$^\sigma$,
restrictedDebits := *cancelledDebits*

70: **upon receive** $\langle$CommitInitState, *e*, *CODState*, *cryptoCDStateCert*$\rangle$ **from** owner q of account *acc*
71: **if** not COD[*acc*][*e* − 1].VerifyCloseStateCert(*CODState*, *cryptoCDStateCert*) **then return**
72: // Check if previously signed a different initial state for COD[*acc*][*e*]
73: **if** *signedStates*[*acc*][*e*] $\neq \bot$ **and** *signedStates*[*acc*][*e*] $\neq$ Hash(*CODState*) **then return**
74: *signedStates*[*acc*][*e*] := Hash(*CODState*)
75: *sigState* := Sign($\langle$CommitInitStateResp, *CODState*$\rangle$)
76: *sigTxs* := Sign($\langle$ConfirmInRecovery, *e* − 1, MerkleTree(*CODState.selectedDebits*).*root*$\rangle$)
77: **send** $\langle$CommitInitStateResp, *sigState*, *sigTxs*$\rangle$ **to** q

Algorithm 4: COD, Global Storage and Account Storage initialization

78: **Storage Objects:**
79: GlobalStorage is an AOStorage object with one key:
key "txs" with initial set of value $\{tx_{init,\ a}\}_{a \in \mathcal{A}}$ and validity function VerifyTxCertForGlobalStorage
80: $\forall a \in \mathcal{A}$: AccountStorage$[a]$ is an AOStorage object with the following keys:
key "debits" with initial set of values $\emptyset$ and validity function VerifyDebit_{acc}
key "state" with initial set of values $\{\langle 1, \langle \{tx_{init,\ a}\}, \emptyset, \emptyset \rangle, \bot \rangle\}$ and validity function $\text{VerifyAccountState}_{acc}$
81: $\forall a \in \mathcal{A} : \text{COD}[a][0] = \text{COD}(a, 1, \emptyset, \{tx_{init,\ a}\}, \emptyset)$

82: **function** VerifyTxCertForGlobalStorage(tx, σ) **returns** Boolean
83: **if** $\langle acc_\sigma, e_\sigma, \sigma_{COD} \rangle = \sigma$ **then**
84: **return** $\text{COD}[acc_\sigma][e_\sigma].\text{VerifyCODCert}(tx, \sigma_{COD})$
85: **return** VerifyRecoveryCert(tx, σ)

86: **function** $\text{VerifyDebit}_{acc}(tx,\ \_)$ **returns** Boolean
87: **return** $tx.sender = acc$ **and** tx is well-formed (including a valid signature)

88: **function** $\text{VerifyAccountState}_{acc}(state,\ \_)$ **returns** Boolean
89: $\langle e, CODState, \sigma_{state} \rangle := state$
90: **return** VerifyTS($\langle$CommitInitStateResp, $nextCODState\rangle, \sigma_{state}$)

2.8.4 Latency breakdown and optimizations

Lines 5, 7 and 10, as well as line 13 on the first iteration of the loop, can all be executed in parallel. This is crucial to achieve the latency claimed in Theorem 2.1.

With this optimization applied and with the instantiation of COD described in Section 2.6, the algorithm achieves latency of 5 round-trips in case of absence of concurrency.

We can further break down the latency costs into 3 categories:

- 2 round-trips are necessary for a consistent broadcast protocol with optimal resilience ($n = 3f + 1$) and a linear number of messages [34]. This is the latency of the purely asynchronous asset transfer protocols with optimal resilience [19, 48], which serve as a baseline for us;
- 2 round-trips to read the up-to-date state (lines 5, 7 and 13 executed in parallel). This facilitates light-weight clients and is especially important in the context of shared accounts, where a client may not always have up-to-date information about its own account;
- 1 round-trip to facilitate the recovery.

As discussed in Section 2.6.2, somewhat surprisingly, the Transfer Concurrency property of CryptoConcurrency does not have an inherent latency cost, as the COD protocol automatically adapts to the current level of contention.

Further optimizations. We believe that one can achieve a smaller latency with any combination of the following techniques (each coming at its own cost):

Use larger quorums: It is a common pattern in distributed computing to trade resilience for latency [5, 6, 83, 94, 99, 108, 125]. Recently, it was exploited in a context very similar to CryptoConcurrency in [125].

Use all-to-all communication between replicas: Naturally, all-to-all communication often offers smaller latency than client-server communication pattern (as in CryptoConcurrency) at the cost of a quadratic number of messages being exchanged. This technique was also adopted by [125].

Execute storage write-back in parallel with COD.Submit: By opening up the storage black box, one may try to perform the write-back phase of ReadKey (described in Section 2.7) in parallel

with the Submit operation of COD. This may reduce the latency by 1 RTT at the cost of slightly complicating the protocol.

Make additional assumptions about the clients: This may help to avoid the need to interact with storage prior to accessing COD. However, relying on the client's local state, even if the client actively listens to all events on the network, would likely require a slight relaxation of the Transfer Consistency property (as defined in Section 2.5).

Make a more monolithic design: In the current design, clients are required to relay information about credits from Global Storage to COD. It may be possible to avoid this if these objects are maintained by the same set of replicas. If combined with the two preceding suggested optimizations, it may be sufficient to avoid the need to read the storage before accessing COD, even without relaxing the consistency requirement.

2.9 Proof Outline

In this section, we sketch the main arguments for Theorem 2.1. In particular, we show that CryptoConcurrency protocol satisfies all the conditions imposed by the theorem. The detailed proof is deferred to Section 2.12.

Transfer Safety. This property requires that, for any account, the balance is always non-negative. This property is ensured by the COD objects used on account acc, which guarantee that, at any time, the set of accepted debit transactions does not surpass the total value of committed credit transactions. We also show that this invariant is preserved during the transition between consecutive COD objects.

Transfer Consistency. To show that our implementation of CryptoConcurrency satisfies the Transfer Consistency property, we define an order on the transactions on a correct account acc and then show that this order is both legal and consistent with the real-time order $\prec_{\mathcal{E},acc}$. More precisely, we order transactions by the epochs they are accepted in, and, then, inside each epoch e, we divide them into three consecutive groups: (i) transactions accepted by COD$[acc][e]$, (ii) selected by Consensus$[acc][e+1]$, and (iii) failed debit transactions canceled by Consensus$[acc][e+1]$. Inside each group we order transactions by $\text{end}_{acc}(tx)$, giving the priority to the credit transactions in case of ties.

Transfer Validity. The proof of Transfer Validity property is relatively simple. We show that CryptoConcurrency satisfies the following two facts: (i) any successful transaction is written to the GlobalStorage, which produces a commit certificate σ_{commit}, and (ii) VerifyCommitCertificate is implemented via verification function of the certificate from the Global Storage.

Transfer Liveness. CryptoConcurrency ensures that every operation invoked by a correct client returns. The proof proceeds as follows. We first show that all operations invoked on Account Storage, Global Storage, and COD eventually terminate. Then, we show that the number of epochs one Transfer operation can span over is finite. Combining these facts, we demonstrate that any operation invoked by a correct client returns.

Account Transactions Completeness. We ensure that CryptoConcurrency satisfies the Account Transactions Completeness property with the help of Global Storage. Every time a client invokes GetAccountTransactions, it essentially reads all committed transactions from GlobalStorage and then filters out the ones that are not relevant.

Transfer Concurrency. In the Transfer Concurrency property, we show that, if from some moment on, there are no overspending attempts observed on a correct account acc (i.e., the total amount spent by all active debit transactions does not exceed the balance at any time t), then there is some moment of time after which no Consensus object is invoked on this account. We prove this by showing that, in such a case, the number of epochs an account goes through is finite, and thus, from some point on, the account does not go through the recovery process and owners do not use Consensus objects.

Latency. To prove that k-overspending-free latency of CryptoConcurrency is $k + 4$ RTTs, we show that the upper bound for the latency of COD.Submit is $k + 1$. Here, k round-trips come from the Prepare phase, and 1 more comes from the Accept phase. We also prove that latencies of AppendKey and ReadKey operations of Append-Only Storage are constant and equal to 2 and 1 RTTs respectively. Finally, we then conclude that with most of the read requests combined as described in Section 2.8.4, we achieve k-overspending-free latency of $k + 4$ RTTs.

2.10 Closable Overspending Detector

In this section, we state the properties Closable Overspending Detector should satisfy and how we implement this abstraction.

2.10.1 Formal definition of Closable Overspending Detector

An instance of COD is identified by an account *acc* and an epoch number e. Additionally, all owners of the account and correct replicas must run the COD instance with the same initial state that consists of:

- $initDebits$: Set$\langle\mathcal{T}\rangle$ – the initial set of debit transactions for account *acc*, $\forall tx \in initDebits$: $tx.sender = acc$;
- $initCredits^\sigma$: Set$\langle$Pair$\langle\mathcal{T}, \Sigma\rangle\rangle$ – initial set of credit transactions for account *acc*, $\forall\langle tx, \sigma\rangle \in initCredits^\sigma$: $tx.recipient = acc$ **and** VerifyCommitCertificate$(tx, \sigma) = true$;
- $restrictedDebits$: Set$\langle\mathcal{T}\rangle$ – the set of debit transactions for account *acc* that this COD object is prohibited from accepting, $restrictedDebits \cap initDebits = \emptyset$;

Moreover, the initial balance of the account must be non-negative (i.e., TotalValue(Txs($initCredits^\sigma$)) $\geq$ TotalValue($initDebits$)).

The Closable Overspending Detector abstraction exports two operations: Submit($debits, credits^\sigma$) and Close($pendingDebits$). It also provides two verification functions: VerifyCODCert(tx, σ) and VerifyCloseStateCert($CODState, \sigma$).

We say that a transaction tx is *accepted by an instance I of COD* iff $tx \in I.initDebits$ or there exists σ such that I.VerifyCODCert$(tx, \sigma) = true$. In order to get new debit transactions accepted, correct clients submit them to the COD object by invoking Submit($debits, credits^\sigma$). Here, $credits^\sigma$ is the set of committed credit transactions the client is aware of with the corresponding commit certificates.

In the most common case when no overspending attempts are detected, Submit($debits, credits^\sigma$) returns OK($debits^\sigma, outCredits^\sigma$), where $debits^\sigma$ contains the same transactions as in *debits* augmented with the cryptographic certificates confirming that these transactions are accepted by a COD and $outCredits^\sigma$ contains enough committed credit transaction to cover all of the transactions in *debits*. More formally, the following two properties are satisfied:

COD-Submit Validity: If a correct client obtains OK($debits^\sigma, outCredits^\sigma$) from Submit($debits, credits^\sigma$), then Txs($debits^\sigma$) = $debits$ and $\forall\langle tx, \sigma\rangle \in debits^\sigma$: VerifyCODCert(tx, σ) = $true$. Moreover, $\forall\langle tx, \sigma\rangle \in outCredits^\sigma$: VerifyCommitCertificate$(tx, \sigma) = true$;

COD-Submit Safety:

- At any time t, the total value of the initial debits ($initDebits$) and the debits accepted by a COD object by time t does not exceed the total value of committed credits on the account *acc* by time t. Moreover, if *acc* is correct, it does not exceed the total amount of all credits returned by the Submit operation by time t;
- No $tx \in restrictedDebits$ is ever accepted by a COD object.

COD.Submit may also return FAIL if the replicas observe an overspending attempt. Note, however, that due to communication delays, a replica may observe only a subset of all the Submit operations that are being executed. Hence, we allow the COD object return FAIL if *any* subset of Submit operations overspends. Another case when we allow the COD object to return FAIL is when some client already invoked the COD.Close operation. More formally:

COD-Submit Success: If (i) *acc* is a correct account, (ii) no client invokes the Close operation, and (iii) for every subset S of invoked operations Submit($debits_i$, $credits_i^\sigma$), TotalValue($debits$) $\leq$ TotalValue($credits$) (where $debits = \bigcup_{i \in S} debits_i \cup initDebits$ and $credits = \text{Txs}(\bigcup_{i \in S} credits_i^\sigma \cup initCredits^\sigma)$, then no Submit operation returns FAIL.

In CryptoConcurrency, once a correct process receives FAIL from an invocation of Submit, it proceeds to closing the COD instance by invoking Close(*pendingDebits*), where *pendingDebits* is an arbitrary set of debit transactions. The operation returns $\langle CODState, \sigma_{state} \rangle$, where *CODState* is a snapshot of the accumulated internal state of the COD object and σ is a certificate confirming the validity of the snapshot. σ_{state} can be later verified by any third party using the VerifyCloseStateCert function.

The snapshot *CODState* contains the following fields:

1. $credits^\sigma$ – the set of credits submitted to this COD along with their commit certificates;
2. *selectedDebits* – a set of debits submitted to this COD such that TotalValue(*selectedDebits*) $\leq$ TotalValue(Txs($credits^\sigma$));
3. *cancelledDebits* – a set of debits submitted to this COD that cannot be added to *selectedDebits* without exceeding the amount of funds provided by $credits^\sigma$.

The set *selectedDebits* must contain all the transactions that have been or ever will be accepted by this COD instance. To this end, as stipulated by the name of the operation, it "closes" the instance of COD and, as already formalized in the COD-Submit Success property, new invocations of the Submit operation may return FAIL even there is no overspending. Formally, operation Close must satisfy the following properties:

COD-Close Validity: If a correct client obtains $\langle CODState, \sigma_{state} \rangle$ from Close(*pendingDebits*), then VerifyCloseStateCert($CODState, \sigma_{state}$) = *true*;

COD-Close Safety: If a correct client obtains $\langle CODState, \sigma_{state} \rangle$ from Close(*pendingDebits*), then:

- $initDebits \subseteq CODState.selectedDebits$ and $restrictedDebits \subseteq CODState.cancelledDebits$;
- for every transaction tx accepted by this COD instance: $tx \in CODState.selectedDebits$;
- $pendingDebits \subseteq CODState.selectedDebits \cup CODState.cancelledDebits$;
- for any Submit operation that returns OK($debits^\sigma$, $outCredits^\sigma$): $outCredits^\sigma \subset CODState.credits^\sigma$ and $\forall \langle tx, \sigma \rangle \in CODState.credits^\sigma$: VerifyCommitCertificate(tx, σ) = *true*. Moreover, $initCredits^\sigma \subseteq CODState.credits^\sigma$;
- TotalValue(Txs($CODState.credits^\sigma$)) $\geq$ TotalValue($CODState.selectedDebits$). Moreover, $\nexists tx \in cancelledDebits$, such that TotalValue(Txs($CODState.credits^\sigma$)) $\geq$ TotalValue($CODState.selectedDebits \cup \{tx\}$);
- $CODState.selectedDebits \cap CODState.cancelledDebits = \emptyset$;

Finally, both operations must eventually terminate:

COD-Liveness: Every call to Submit and Close operations by a correct client eventually returns.

In CryptoConcurrency, each account *acc* is provided with a list of COD objects COD$[acc][1, 2, \ldots]$ and for all $e \geq 1$, the initial state (*initDebits*, $initCredits^\sigma$, and *restrictedDebits*) of COD$[acc][e+1]$ is initialized using a snapshot returned by COD$[acc][e]$ (*selectedDebits*, $credits^\sigma$, and *cancelledDebits*).

2.10.2 Implementation of Closable Overspending Detector

The variables defining a Closable Overspending Detector object and a replica's state are listed in Algorithm 2.5. The pseudocode of the verifying functions is provided Algorithm 2.6, while the protocols of a client and a replica are presented in Algorithms 2.7 and 2.8, respectively. Here we implicitly assume that each protocol message and each message being signed carries information on the account *acc* and the epoch the COD object is parameterized with. This ensures that different instances of COD do not interfere with each other and, in particular, that signatures created in one instance cannot be used in another one.

The Submit operation consists of two phases: Prepare and Accept.

Prepare phase. The implementation of the Prepare phase inherits the key ideas from the Generalized Lattice Agreement protocol of [61]. In this phase, each debit transaction is appended with a set of *dependencies*, i.e., credit transactions that are submitted to COD along with the debit transaction, signed by the client (line 131). Intuitively, this is necessary in order to prevent Byzantine replicas from falsely detecting overspending simply by ignoring credit transactions. Then, the client sends a PREPARE message containing all the debit transactions (with signed sets of dependencies attached to them) and credit transactions (with commit certificates) it is aware of to all the replicas (line 133) and waits for their replies (line 134). It also attaches the set of debit transactions that it started the Prepare phase with (*submitDebits*). If any replica replies with any debit transactions the client is unaware of, the client repeats the request with the updated sets of debits and credits (line 144).

There are multiple ways in which this loop may terminate. In a successful scenario, the client either manages to collect a quorum of signed replies with an identical set of debits (line 141) or receives a notification that some other client already managed to prepare a set of transactions that includes all the transactions from *submitDebits* (line 136). In these two cases, the client returns will move on to the Accept phase with a quorum of signatures as a certificate that it performed the Prepare phase correctly (lines 137 and 143).

If the total value of all debits that the client is aware of (including the ones from ongoing Submit operations) exceeds the total value of committed credits known to the client (line 139), this indicates a potential overspending attempt. As a result, the client will return FAIL both from the Prepare phase (line 140) and from the Submit operation (line 125). Finally, if the client is notified (with a valid signature from another owner of the account) that this COD instance is being closed, it also returns FAIL (line 135).

When a correct replica receives a PREPARE message from a client, it first checks that the COD instance is not yet closed (line 164) and that the transactions in the *submitDebits* set are not yet prepared (line 165). If one of these conditions does not hold, the replica notifies the client with a proper certificate and stops processing the message. Otherwise, the replica proceeds to check that the client's message is well-formed (lines 168 to 173) and, if it is, the replica adds the received transactions to its local state (lines 175 and 176). Finally, the replica sends to the client all debit transactions (with signed dependencies) and all credit transactions (with commit certificates) it is aware of. It also attaches a digital signature on the set of debits if the account's balance is non-negative after all the transactions the replica is aware of are applied (lines 176 to 179). Intuitively, the signature indicates that the replica acknowledged these debits. As described above, if a client collects a quorum of such signatures for an identical set of debits, it can move on to the Accept phase.

Accept phase. In the Accept phase (lines 145 to 152), the client gathers signatures for the Merkle tree root of the set of debits it obtained during the Prepare phase. Once it has collected signatures from a quorum of replicas, the client constructs certificates for the debit transactions with which it invoked the Submit operation and returns them along with all credits used to cover these transactions. The client may return FAIL in the Accept phase if another client has invoked the Close operation.

The purpose of the Accept phase is, intuitively, to ensure that each transaction accepted by this COD instance is stored in the *preparedDebits* set on at least a quorum of replicas. This is necessary to guarantee that, in the Close operation, the clients will be able to reliably identify which transactions could have been accepted (by looking at the *preparedDebits* sets reported by the replicas).

The Close operation. The Close operation is designed to deactivate the COD object and to collect a snapshot of its state. During this operation, the client collects the states of a quorum of replicas and then asks the replicas to sign the accumulated joint state. Accessing a quorum guarantees that the client gathers all the debits that have been previously returned from the Submit operations. As a result, confirmed transactions are never lost.

The interaction between the ACCEPTREQUEST and CLOSE messages in the COD implementation is similar to that of "propose" messages and "prepare" messages with a larger ballot number in Paxos [98]. Intuitively, it guarantees that, if there is a Submit concurrent with a Close, either the client executing Submit will "see" the Close operation (i.e., will receive a CLOSED message) and will return FAIL or the client executing the Close operation will "see" the debits in a replica's *preparedDebits* set.

Algorithm 2.5 Closable Overspending Detector (Parameters and replica state)

Parameters:
- *acc* – account for which this COD is used
- *e* – epoch number of this COD
- *initDebits* – set of initial debits used by this COD
- $initCredits^{\sigma}$ – set of initial credits used by this COD
- *restrictedDebits* – set of debits that should not be accepted by this COD

Replica State:
- $debits^{\sigma}$ – the set of debits acknowledged by this replica, initially $\{\langle tx, \bot\rangle \mid tx \in initDebits\}$
- $credits^{\sigma}$ – the set of known credits, initially $initCredits^{\sigma}$
- *preparedDebits* – prepared debits received in AcceptRequest messages, initially $\emptyset$
- $\sigma_{prepare}$ – certificate for *preparedDebits*, initially $\bot$
- *isClosed* – current status of this COD, initially *false*
- σ_{closed} – proof of the fact that COD was closed, initially $\bot$

Algorithm 2.6 COD verifying and helper functions

```
public function VerifyCODCert(tx, σ)
    ⟨root, itemProof, σ_MT, acc_σ, e_σ⟩ := σ
    if acc_σ ≠ acc or e_σ ≠ e then return false
    return VerifyItemProof(root, itemProof, tx) and VerifyTS(⟨AcceptAck, root⟩, σ_MT)

public function VerifyCloseStateCert(closedState, closedStateCert) returns Boolean
    let ⟨credits^σ, selectedDebits, cancelledDebits⟩ be closedState
    if credits^σ are not sufficient to cover all debits in selectedDebits then return false
    return VerifyTS(⟨ConfirmStateResp, selectedDebits, cancelledDebits⟩, closedStateCert)

function SplitDebits(pendingDebits, messages) returns Pair⟨Set⟨T⟩, Set⟨T⟩⟩
    assert messages is a set of tuples {⟨CloseResp, credits_i^σ, preparedDebits_i, σ_prepare_i, sig_i⟩}_{i∈Q}
    let preparedDebits := ⋃_{i∈Q} preparedDebits_i
    let allCredits := Txs(initCredits^σ) ∪ (⋃_{i∈Q} Txs(credits_i^σ))
    assert TotalValue(allCredits) ≥ TotalValue(initDebits ∪ preparedDebits)
    // At least all initial debits and all prepared debits must be selected
    selectedDebits := initDebits ∪ preparedDebits
    for tx ∈ pendingDebits \ restrictedDebits do
        if TotalValue(selectedDebits ∪ {tx}) ≤ TotalValue(allCredits) then
            selectedDebits := selectedDebits ∪ {tx}
    return ⟨selectedDebits, restrictedDebits ∪ (pendingDebits \ selectedDebits)⟩
```

Algorithm 2.7 COD (code for client p)

123: **operation** Submit($debits, credits^\sigma$) **returns** OK(Pair⟨Set⟨$\mathcal{T}_\Sigma$⟩, Set⟨$\mathcal{T}_\Sigma$⟩⟩) or FAIL
124: $prepareResult$ = Prepare($debits, credits^\sigma$)
125: **if** $prepareResult$ **is** FAIL **then return** FAIL
126: let OK(⟨$preparedDebits, allCredits^\sigma, \sigma_{prepare}$⟩) be $prepareResult$
127: **return** Accept($debits, preparedDebits, allCredits^\sigma, \sigma_{prepare}$)

128: **function** Prepare($debits$, $credits^\sigma$) **returns** OK(Tuple⟨Set⟨$\mathcal{T}$⟩, Set⟨$\mathcal{T}_\Sigma$⟩, Σ⟩) or FAIL
129: $submitDebits := debits$
130: $deps := \{tx.id \mid tx \in \text{Txs}(credits^\sigma)\}$
131: $debits^\sigma := \{\langle tx, \langle deps, \text{Sign}(\langle \textsc{Deps}, tx, deps\rangle)\rangle\rangle \mid tx \in debits\}$
132: **while** *true* **do**
133: **send** ⟨PREPARE, $debits^\sigma$, $credits^\sigma$, $submitDebits$⟩ **to** all replicas
134: **wait for** quorum Q of valid PREPARERESP replies ⟨PREPARERESP, $debits_i^\sigma, credits_i^\sigma, sig_i$⟩
or 1 valid ALREADYPREPARED reply or 1 valid CLOSED reply
135: **if** received a valid ⟨CLOSED, sig⟩ reply **then return** FAIL
136: **if** received a valid ⟨ALREADYPREPARED, $preparedDebits, \sigma_{prepare}$⟩ reply **then**
137: **return** ⟨$preparedDebits, \sigma_{prepare}$⟩ // Someone has already prepared a set with our transactions.
138: $allDebits^\sigma := debits^\sigma \cup \left(\bigcup_{i\in Q} debits_i^\sigma\right)$; $allCredits^\sigma := credits^\sigma \cup \left(\bigcup_{i\in Q} credits_i^\sigma\right)$
139: **if** balance(Txs($allDebits^\sigma$) ∪ Txs($allCredits^\sigma$), acc) < 0 **then**
140: **return** FAIL // Impossible to add all transactions without violating the invariant.
141: **else if** $\forall i \in Q$: Txs($debits_i^\sigma$) = Txs($allDebits^\sigma$) **and** $sig_i \neq \perp$ **then**
142: $\sigma_{prepare}$:= CreateTS(⟨PREPARERESP, Txs($allDebits^\sigma$)⟩, $\{sig_i\}_{i\in Q}$)
143: **return** OK(⟨Txs($allDebits^\sigma$), $allCredits^\sigma, \sigma_{prepare}$⟩)
144: $debits^\sigma := allDebits^\sigma$; $credits^\sigma := allCredits^\sigma$

145: **function** Accept($submittedDebits$, $preparedDebits$, $allCredits^\sigma$, $\sigma_{prepare}$) **returns** OK(Pair⟨Set⟨$\mathcal{T}_\Sigma$⟩, Set⟨$\mathcal{T}_\Sigma$⟩⟩) or FAIL
146: **send** ⟨ACCEPTREQUEST, $preparedDebits$, $allCredits^\sigma$, $\sigma_{prepare}$⟩ **to** all replicas
147: **wait for** valid ⟨ACCEPTACK, sig_i⟩ replies from a quorum Q or 1 valid CLOSED reply
148: **if** received a valid ⟨CLOSED, sig⟩ reply **then return** FAIL
149: $merkleTree$ = MerkleTree($preparedDebits$)
150: σ_{MT} = CreateTS(⟨ACCEPTACK, $merkleTree.root$⟩, $\{sig_i\}_{i\in Q}$)
151: $submittedDebits^\sigma = \{\langle tx, \langle merkleTree.root, \text{GetItemProof}(merkleTree, tx), \sigma_{MT}, acc, e\rangle\rangle \mid tx \in submittedDebits\}$
152: **return** OK($submittedDebits^\sigma$, $allCredits^\sigma$)

153: **operation** Close($pendingDebits$) **returns** Pair⟨CloseState, Σ⟩
154: **send** ⟨CLOSE, Sign(⟨CLOSE, e⟩)⟩ **to** all replicas
155: **wait for** a quorum Q of valid CLOSERESP replies
156: $messages$:= replies ⟨CLOSERESP, $credits_i^\sigma, preparedDebits_i, \sigma_{prepare_i}, sig_i$⟩ from quorum Q
157: $credits^\sigma := \bigcup credits_i^\sigma$
158: ⟨$selectedDebits, cancelledDebits$⟩ := SplitDebits($pendingDebits, messages$)
159: **send** ⟨CONFIRMSTATE, $pendingDebits$,$messages$⟩ **to** all replicas
160: **wait for** a quorum Q of valid ⟨CONFIRMSTATERESP, sig_i⟩ replies
161: $closedStateCert$:= CreateTS(⟨CONFIRMSTATERESP, $selectedDebits, cancelledDebits$⟩, $\{sig_i\}_{i\in Q}$)
162: **return** ⟨⟨$credits^\sigma, selectedDebits, cancelledDebits$⟩, $closedStateCert$⟩

Algorithm 2.8 COD (code for replica r)

163: **upon receive** $\langle$PREPARE, $receivedDebits^\sigma$, $receivedCredits^\sigma$, $submitDebits\rangle$ **from** client q
164: **if** $isClosed$ **then send** $\langle$CLOSED, $\sigma_{closed}\rangle$ **to** q and **return**
165: **if** $submitDebits \subseteq preparedDebits$ **then**
166: **send** $\langle$ALREADYPREPARED, $preparedDebits$, $\sigma_{prepare}\rangle$ **to** q
167: **return**
168: **for** all $tx \in$ Txs($receivedDebits^\sigma$) **do**
169: **if** $tx.sender \neq acc$ **then return**
170: **if** $tx \in restrictedDebits$ **then return**
171: **for** all $\langle tx, \sigma_{credit}\rangle \in receivedCredits^\sigma$ **do**
172: **if** $tx.recipient \neq acc$ **or** not VerifyCommitCertificate(tx, σ_{credit}) **then return**
173: **if** $(\bigcup_{\langle tx,\langle deps,\sigma\rangle\rangle \in receivedDebits^\sigma} deps) \not\subseteq$ Txs($receivedCredits^\sigma$) **then return**
174: $credits^\sigma := credits^\sigma \cup receivedCredits^\sigma$
175: $debits^\sigma := debits^\sigma \cup receivedDebits^\sigma$
176: **if** balance(Txs($debits^\sigma \cup credits^\sigma$), acc) ≥ 0 **then**
177: $sig :=$ Sign($\langle$PREPARERESP, Txs($debits^\sigma$)$\rangle$)
178: **else** $sig := \bot$
179: **send** $\langle$PREPARERESP, $debits^\sigma$, $credits^\sigma$, $sig\rangle$ **to** q

180: **upon receive** $\langle$ACCEPTREQUEST, $receivedDebits$, $receivedCredits^\sigma$, $\sigma\rangle$ **from** client q
181: **if** $isClosed$ **then send** $\langle$CLOSED, $\sigma_{closed}\rangle$ **to** q and **return**
182: **if** not VerifyTS($\langle$PREPARERESP, $receivedDebits\rangle, \sigma$) **then return**
183: **for** all $\langle tx, \sigma\rangle \in receivedCredits^\sigma$ **do**
184: **if** not VerifyCommitCertificate(tx, σ) **then return**
185: $credits^\sigma := credits^\sigma \cup receivedCredits^\sigma$
186: **if** $preparedDebits \subset debits$ **then**
187: $preparedDebits := debits$; $\sigma_{prepare} := \sigma$
188: $sig :=$ Sign($\langle$ACCEPTACK, MerkleTree($debits$).$root\rangle$)
189: **send** $\langle$ACCEPTACK, $sig\rangle$ **to** q

190: **upon receive** $\langle$CLOSE, $sig\rangle$ **from** client q
191: **if** not VerifySignature($\langle$CLOSE, $e\rangle, sig, q$) **then return**
192: $isClosed := true$; $\sigma_{closed} := sig$
193: $sig :=$ Sign($\langle$CLOSERESP, $preparedDebits\rangle$)
194: **send** $\langle$CLOSERESP, $credits^\sigma$, $preparedDebits, \sigma_{prepare}$, $sig\rangle$ **to** q

195: **upon receive** $\langle$CONFIRMSTATE, $pendingDebits$, $messages\rangle$ **from** client q
196: **if** $messages$ contains invalid messages **then return**
197: $\langle selectedDebits, cancelledDebits\rangle :=$ SplitDebits($pendingDebits, messages$)
198: $sig :=$ Sign($\langle$CONFIRMSTATERESP, $selectedDebits$, $cancelledDebits\rangle$)
199: **send** $\langle$CONFIRMSTATERESP, $sig\rangle$ **to** client q

2.11 Append-Only Storage

First, we give a formal definition for Append-Only Storage in 2.11.1 and then proceed with its detailed implementation in 2.11.2.

2.11.1 Formal definition of Append-Only Storage

The abstraction is parameterized with a set of tuples $\{\langle k_1, vs_{init_{k_1}}, \text{VerifyInput}_{k_1}\rangle, \ldots, \langle k_n, vs_{init_{k_n}}, \text{VerifyInput}_{k_n}\rangle\}$. Each tuple consists of a key k_i, initial set of values $vs_{init_{k_i}}$ for this key, and a verifying function for this key VerifyInput_{k_i}, which takes a value v and a certificate σ_v for this value and returns *true* if $\langle v, \sigma_v\rangle$ is valid input for key k_i, and *false* otherwise.[4] Also, any initial value for a given key is valid: $\forall v \in vs_{init_{k_i}} : \text{VerifyInput}_{k_i}(k_i, v, \perp) = true$.

The abstraction exports two operations: $\text{AppendKey}(k, v, \sigma_v)$ and $\text{ReadKey}(k)$. It also provides one boolean function: $\text{VerifyStoredCert}(k, v, \sigma_{AOS})$.

The operation $\text{AppendKey}(k, v, \sigma_v)$ accepts a key k and a value v together with its certificate σ_v and adds v to the set of stored values for k in Append-Only Storage, but only if $\text{VerifyInput}_k(v, \sigma_v) = true$. As a result, the $\text{AppendKey}(k, v, \sigma_v)$ operation outputs a certificate σ_{AOS}, which is an evidence of the fact that value v is stored in the set corresponding to the key k in Append-Only Storage.

The $\text{ReadKey}(k)$ operation returns vs^{σ}_{AOS}, where vs^{σ}_{AOS} is a set of pairs $\langle v, \sigma_{AOS}\rangle$, such that: v is a valid value for a key k (i.e., there exists a certificate σ_v, such that $\text{VerifyInput}_k(v, \sigma_v) = true$), and σ_{AOS} is a certificate that proves that v belongs to the set of values for the key k in the Append-Only Storage.

Let us formally define the properties of Append-Only Storage:

AOS-Consistency: If there exists σ_{AOS} such that $\text{VerifyStoredCert}(k, v, \sigma_{AOS}) = true$ at the moment when $\text{ReadKey}(k)$ was invoked by a correct client, then the output of this operation will contain v (paired with a certificate);

AOS-Input Validity: If there exists a certificate σ_{AOS} such that $\text{VerifyStoredCert}(k, v, \sigma_{AOS}) = true$, then there exists σ_v, such that $\text{VerifyInput}_k(v, \sigma_v)$;

AOS-Output Validity: If a correct client returns σ_{AOS} from $\text{AppendKey}(k, v, \sigma_v)$, then $\text{VerifyStoredCert}(k, v, \sigma_{AOS}) = true$. Moreover, if a correct client returns vs^{σ}_{AOS} from $\text{ReadKey}(k)$, then $\forall \langle v, \sigma_{AOS}\rangle \in vs^{\sigma}_{AOS} : \text{VerifyStoredCert}(k, v, \sigma_{AOS}) = true$;

AOS-Liveness: All operations eventually terminate.

2.11.2 Implementation of Append-Only Storage

The pseudocode for the Append-Only Storage can be found in Algorithm 2.9.

In operation $\text{AppendKey}(k, v, \sigma_v)$, the client calls function WriteValuesToKey with a key k and a singleton set $\{\langle v, \sigma_v\rangle\}$ as parameters. In WriteValuesToKey, given a key k and a set of values with certificates vs^{σ}, the client first sends k and vs^{σ} to all replicas and waits for a quorum of valid replies. The client then returns the submitted values, provided with their aggregated Merkle Tree signatures, and extracted certificates for all v in vs^{σ} ($\langle v, * \rangle \in vs^{\sigma}$).

In operation $\text{AppendKey}(k, v, \sigma_v)$, the call to WriteValuesToKey returns a set consisting of only one pair $\langle v, \sigma_{AOS}\rangle$ (due to the input being a singleton set). Finally, AppendKey returns σ_{AOS}.

In the $\text{ReadKey}(k)$ operation, the client first requests the values stored by a quorum of replicas for this key. The aggregated set of values vs^{σ} is then passed to the WriteValuesToKey function to make sure that any value v in vs^{σ} is written to a quorum of processes. This guarantees that any value read from the Append-Only Storage will be read from it again later. Finally, vs^{σ} is returned by the operation.

In CryptoConcurrency, we use two types of the Append-Only Storage. The first is Global Storage which allows different accounts to interact with each other (i.e., receive incoming transactions). The other one – Account Storage is used per account, i.e., only clients that share an account can communicate with it and every account has an Account Storage associated with it.

As there is one Global Storage per system, it makes sense to implement it on the same set of replicas as all of the other parts of the algorithm. At the same time, Account Storage serves only one account, and, in fact, can be implemented on a different, *local* set of replicas for each account.

[4] Sometimes, we define the validity based only on the values themselves and use $\perp$ for the certificates.

Algorithm 2.9 Append-Only Storage

```
    // Code for client p
200: operation ReadKey(k) returns Set⟨Pair⟨V, Σ⟩⟩
201:     send ⟨ReadKey, k⟩ to all replicas
202:     wait for valid ⟨ReadKeyResp, vs_i^σ⟩ replies from a quorum Q
203:     vs^σ := ⋃_i vs_i^σ
204:     return WriteValuesToKey(vs^σ)

205: operation AppendKey(k, v, σ_v) returns Σ
206:     {⟨v, σ_AOS⟩} := WriteValuesToKey(k, {⟨v, σ_v⟩})
207:     return σ_AOS

208: operation WriteValuesToKey(k, vs^σ) returns Set⟨Pair⟨V, Σ⟩⟩
209:     send ⟨AppendKey, k, vs^σ⟩ to all replicas
210:     wait for valid ⟨AppendKeyResp, sig_i⟩ replies from a quorum Q
211:     vs := {v | ⟨v, σ_v⟩ ∈ vs^σ}
212:     merkleTree := MerkleTree(vs)
213:     σ_MT := CreateTS(⟨AppendKeyResp, k, merkleTree.root⟩, {sig_i}_{i∈Q})
214:     return {⟨v, ⟨merkleTree.root, GetItemProof(merkleTree, v), σ_MT⟩⟩ | v ∈ vs}

215: public function VerifyStoredCert(k, v, σ_AOS) returns Boolean
216:     ⟨root, itemProof, σ_MT⟩ := σ_AOS
217:     return VerifyItemProof(root, itemProof, v) and VerifyTS(⟨AppendKeyResp, k, root⟩, σ_MT)

    // Code for replica r
218: State:
219:     log – mapping from a key to a set of values for this key, initially ∀k : log[k] = ∅
220: upon receive ⟨ReadKey, k⟩ from client p
221:     send ⟨ReadKeyResp, log[k]⟩ to p

222: upon receive ⟨AppendKey, k, vs^σ⟩ from client p
223:     for all ⟨v, σ_v⟩ ∈ vs^σ do
224:         if not VerifyInput_k(v, σ_v) then return
225:     log[k] := log[k] ∪ vs^σ
226:     vs := {v | ⟨v, σ_v⟩ ∈ vs^σ}
227:     sig := Sign(⟨AppendKeyResp, k, MerkleTree(vs).root⟩)
228:     send ⟨AppendKeyResp, sig⟩ to p
```

2.12 Proofs of correctness

In 2.12.1 we prove the correctness of the CryptoConcurrency protocol, assuming the correctness of the underlying building blocks. Then, in 2.12.2 and 2.12.3, we show that the implementations of Closable Overspending Detector and Append-Only Storage are correct.

2.12.1 Proof of Correctness: CryptoConcurrency

In this subsection, we prove that CryptoConcurrency satisfies the six Asset Transfer properties Transfer Liveness, Transfer Validity, Transfer Safety, Transfer Consistency, Account Transactions Completeness and Transfer Concurrency, as defined in Section 2.5.

Transfer Liveness. Let us start with the proof of the Transfer Liveness property. First of all, it is important to note that all invocations of operations of COD, Append-Only Storage (Account Storage and Global Storage), and Consensus will eventually terminate due to the liveness properties of these objects (namely, COD-Liveness, AOS-Liveness and C-Liveness). Moreover, in the implementation of CryptoConcurrency presented in Algorithm 2.1, there is just one "**wait for** " statement (line 51) and one potentially infinite loop (lines 11 to 34). Hence, we need to prove their eventual termination.

Lemma 2.2. *Line 51 invoked by a correct client always eventually terminates.*

Proof. Let us say that two CommitInitState messages are *conflicting* if they contain the same epoch, but different *nextCODState* fields. By inspecting lines 70 to 77, it is easy to verify that, unless the owners of some account issue conflicting CommitInitState messages, the correct replicas will reply to each CommitInitState message that contains a *CODState* with a valid certificate.

Thanks to the C-Consistency property of consensus, correct owners of the same account will never send conflicting CommitInitState messages. Moreover, according to our assumptions, correct clients never share their account with Byzantine clients (see Section 2.3.2).

Hence, a correct client that reached line 51 will always eventually collect a quorum of replies and will move on to the next line. □

Lemma 2.3. *A correct client p executing Transfer(tx^*) enters the loop of lines 11 to 34 a finite number of times.*

Proof. Let *acc* be the account of client *p*.

Let us consider the moment t when p returns from the invocation on line 10. Let us consider all clients that are executing the Transfer operation on *acc* at time t. Let e_{max} be the maximum of their epoch numbers.

By the AOS-Consistency property of Account Storage, whenever any owner of *acc* enters epoch $e_{max}+1$, it will have tx^* in its variable *pendingDebits* on line 15. This implies that, whenever any process invokes Recovery($e_{max}+1$, *pendingDebits*) on line 26, $tx^* \in$ *pendingDebits*. By the COD-Close Safety property of COD, tx^* will belong to the *CODState* (either to *CODState.selectedDebits* or *CODState.cancelledDebits*) state received by any owner of *acc* invoking COD[*acc*][$e_{max}+1$].Close(*pendingDebits*) on line 44. Finally, by the C-Validity property of Consensus, tx^* will also belong to *nextCODState* on line 47.

Since with each iteration of the loop p increments its own epoch number, it will either eventually exit the loop or reach line 27 with $e = e_{max} + 1$ and, as we just established, will find tx^* in either *CODState.selectedDebits* or *CODState.cancelledDebits*. In any case, the client will terminate Transfer(tx^*) with either OK(σ_{commit}) on line 30 or FAIL on line 32.

□

Theorem 2.4. *CryptoConcurrency satisfies the Transfer Liveness property of Asset Transfer.*

Proof. Liveness of Transfer operation follows directly from Theorems 2.2 and 2.3 and the liveness properties of the underlying building blocks. Liveness of GetAccountTransactions operation follows from the liveness property of Append-Only Storage (AOS-Liveness). □

Transfer Validity. Now we proceed by proving the Transfer Validity property.

Theorem 2.5. *CryptoConcurrency satisfies the Transfer Validity property of Asset Transfer.*

Proof. This theorem follows from the implementation of the algorithm and the AOS-Output Validity property of the Append-Only Storage. If a correct client returns OK(σ_{commit}) from the Transfer(tx) operation, then it successfully returned σ_{commit} from GLOBALSTORAGE.AppendKey($'txs', tx, \sigma$) (at either line 23 or line 29). As the implementation of VerifyCommitCertificate(tx, σ_{commit}) is essentially GLOBALSTORAGE.VerifyStoredCert($'txs', tx, \sigma_{commit}$), by the AOS-Output Validity property VerifyCommitCertificate(tx, σ_{commit}) = *true*. □

Transfer Safety. The Transfer Safety is a type of property that is essential for any asset transfer system. It tells us that no account can overspend. We now show that CryptoConcurrency satisfies this property.

For the proof, let us consider an account *acc*. We say that a debit transaction tx on *acc* is *associated with an epoch* e, iff e is the minimum epoch number such that:

- Either there exists a certificate σ_{COD} such that COD[*acc*][e].VerifyCODCert(tx, σ_{COD}) = *true*;
- Or there exists $\sigma_{recovery} = \langle *, *, *, e \rangle$ such that VerifyRecoveryCert($tx, \sigma_{recovery}$) = *true*.

Lemma 2.6. *For any epoch number e, COD[acc][e].initDebits includes all committed debit transactions associated with any epoch $e' < e$.*

Proof. We prove this lemma by induction. The base case of the induction ($e = 1$) is trivially satisfied as there are no debit transactions associated with epoch numbers less than 1 (note that genesis transactions are credits).

Now, assuming that the statement of this lemma holds up to the epoch number e, we prove that it also holds for the epoch number $e + 1$. The initial state of the COD object for epoch $e + 1$ (in particular, COD[*acc*][$e + 1$].*initDebits*) is formed from the output value $\langle nextCODState, * \rangle$ of the CONSENSUS[*acc*][$e + 1$].Propose operation. Note that *nextCODState* is also an input of CONSENSUS[*acc*][$e + 1$] and it must satisfy COD-Close Safety property to be accepted by a quorum of replicas (i.e., pass the check at line 71). Particularly, any transaction accepted by COD[*acc*][e] is in *nextCODState.selectedDebits*. Furthermore, by the implementation of the Recovery function, for any transaction tx, such that VerifyRecoveryCert($tx, \sigma_{recovery}$) = *true* and $\sigma_{recovery} = \langle *, *, *, e \rangle$, the following holds: $tx \in$ *nextCODState.selectedDebits*. This means that any transaction associated with epoch e is in COD[*acc*][$e + 1$].*initDebits*. The fact that any transaction associated with epoch $e' < e$ is in COD[*acc*][$e+1$].*initDebits* follows from the part of the COD-Close Safety property of the COD abstraction saying that COD[*acc*][e].*initDebits* $\subseteq$ *nextCODState.selectedDebits*. By induction we know that any committed transaction associated with epoch $e' < e$ is in COD[*acc*][e].*initDebits*. Thus, if the statement of this lemma holds up to an epoch number e, then it also holds up to an epoch number $e + 1$, which concludes the induction.

□

We say that account *acc* is in epoch e at a given moment in time iff, at this moment, there exists a correct replica that initialized COD[*acc*][e] object at line 69, but no correct replica initialized COD[*acc*][$e + 1$] yet.

Theorem 2.7. *CryptoConcurrency satisfies the Transfer Safety property of Asset Transfer.*

Proof. We prove this theorem by contradiction. Let us assume that CryptoConcurrency does not satisfy Transfer Safety property, i.e., there exists an account *acc*, such that, at some moment of time t, balance($C(t), acc$) < 0. Let us consider the first moment of time t_0 when it happens and an epoch e account *acc* is in at time t_0. We know that a transaction can be committed if it obtains certificate via a COD object or via the Recovery procedure. From the COD-Submit Safety property of COD object, we know that the for any time t total value of COD[*acc*][e].*initDebits* and debits accepted by a COD object by time t for an account *acc* does not exceed total value of committed credits on account *acc* by time t. By Lemma 2.6, we know that COD[*acc*][e].*initDebits* includes all committed debit transactions associated with any epoch $e' < e$. Also, let us consider a

set of committed transactions *selectedDebits* that are associated with an epoch $e' \leq e$ and such that $\forall tx \in debits$: VerifyRecoveryCert$(tx, e, \sigma) = true$. By the implementation, there exists a quorum of processes that signed a message $\langle$CommitInitStateResp, $\langle selectedDebits, credits^{\sigma}\, cancelledDebits\rangle\rangle$. Then, $\langle selectedDebits, credits^{\sigma}, cancelledDebits\rangle$ should satisfy COD-Close Safety property, in particular TotalValue($selectedDebits$) $\geq$ TotalValue(Txs($credits^{\sigma}$)). Also, note that $credits^{\sigma}$ is a set of committed credits on account acc and Txs($credits^{\sigma}$) $\subseteq$ credits($C(t_0), acc$). In addition, from Lemma 2.6 and the COD-Close Safety property of COD, we know that *selectedDebits* includes all transactions that have been accepted by COD$[acc][e]$ and all committed transactions associated with an epoch $e' < e$. This implies that balance($C(t_0), acc$) ≥ 0, which contradicts our assumption. Consequently, for all t and for all acc balance($C(t), acc$) ≥ 0.

□

Transfer Consistency. Next, we prove that CryptoConcurrency satisfies the Transfer Consistency property.

Let us briefly outline the proof structure. Consider any execution $\mathcal{E}$. We need to show that there exists a legal permutation of transactions in $\mathcal{T}(\mathcal{E}, acc)$ that is consistent with $\prec_{\mathcal{E},acc}$ for a correct account acc. First, we provide some formalism that we will use during the proof. Then, we construct a permutation of transactions in $\mathcal{T}(\mathcal{E}, acc)$. Using given definitions, by induction on the epoch number, we show that the constructed permutation is legal and consistent with $\prec_{\mathcal{E},acc}$.

Given a correct account acc, we say that n is the *final epoch for acc in* $\mathcal{E}$ iff this is the largest number such that at least one correct replica initialized COD$[acc][n]$ at line 69. Any epoch with a smaller number is said to be *non-final*.

For the rest of this proof section, we consider a correct account acc.

We say that transaction tx *belongs* to epoch n iff it belongs to one of the following three groups:

- Group $G_1(n)$:
  - debit transactions accepted by COD$[acc][n]$, i.e., every tx, such that there exists COD$[acc][n]$.Submit(...) that returned OK($acceptedDebits^{\sigma}, *$), such that $\langle tx, * \rangle \in acceptedDebits^{\sigma}$, excluding the transactions from COD$[acc][n].initDebits$;
  - credit transactions returned from COD$[acc][n]$, i.e., any tx such that there exists COD$[acc][n]$.Submit(...) that returned OK($*, credits^{\sigma}$), such that $\langle tx, * \rangle \in credits^{\sigma}$, excluding the transactions from COD$[acc][n].initCredits^{\sigma}$;
- Group $G_2(n)$:
  - debit and credit transactions selected by Consensus$[acc][n+1]$, i.e., any transaction tx, such that Consensus$[acc][n+1]$.Propose(...) returns $\langle\langle allCredits^{\sigma}, selectedDebits, cancelledDebits\rangle, *\rangle$ and $tx \in selectedDebits$ or $\langle tx, * \rangle \in allCredits^{\sigma}$, excluding the transactions from $G_1(n)$, COD$[acc][n].initDebits$ and COD$[acc][n].initCredits^{\sigma}$;
- Group $G_3(n)$:
  - debit transactions canceled by Consensus$[acc][n+1]$, i.e., any debit transaction tx, such that Consensus$[acc][n+1]$.Propose(...) returns $\langle\langle allCredits^{\sigma}, selectedDebits, cancelledDebits\rangle, *\rangle$ and $tx \in cancelledDebits$, excluding COD$[acc][n].restrictedDebits$.

Let $E(n)$ denote the set of transactions that belong to epoch n (i.e., $E(n) = G_1(n) \cup G_2(n) \cup G_3(n)$). Each transaction *belongs* to at most one epoch, i.e., $\forall n_1 \neq n_2 : E(n_1) \cap E(n_2) = \emptyset$. Also, for any committed transaction tx with $acc \in \{tx.sender, tx.recipient\}$ $\exists n \geq 0$, for which $tx \in E(n)$.

For completeness, we also define $E(0)$ as a set that consists of only one special group $G_1(0)$, which contains only genesis transaction $tx_{init,\ acc}$.

Lemma 2.8. *Transaction tx belongs to one of the sets COD$[acc][n].initDebits$, Txs(COD$[acc][n].initCredits^{\sigma}$), or COD$[acc][n].restrictedDebits$ iff $tx \in E(n')$ for some $n' < n$.*

Proof. The implication from left to right follows by induction from the definition of groups G_2 and G_3 and the implementation of the Recovery procedure.

Similarly, the other direction follows by induction from the COD-Close Safety property and the implementation of the Recovery procedure, in a way analogous to Theorem 2.6. □

Lemma 2.9. *Each transaction tx on account acc belongs to exactly one of the groups, i.e., $\forall tx \in \mathcal{T}(\mathcal{E}, acc) : \exists$ unique pair (n, i) such that $tx \in G_i(n)$.*

Proof. Follows from Theorem 2.8 and the definition of groups. □

Let us now define a total order $<$ on the transactions in $\mathcal{T}(\mathcal{E}, acc)$ as follows:

(i) First, we order the transactions by their epoch numbers, i.e.: $\forall k, m$ s.t. $k \neq m : \forall tx \in E(k), tx' \in E(m)$: $tx < tx'$ iff $k < m$;

(ii) Within an epoch, by their group numbers: $\forall n, i, j$ s.t. $i \neq j : \forall tx \in G_i(n), tx' \in G_j(n)$: $tx < tx'$ iff $i < j$;

(iii) Within each group, by end_{acc}, giving the priority to the credit transactions: $\forall n, i : \forall tx, tx' \in G_i(n) : tx < tx'$ iff $\text{end}_{acc}(tx) < \text{end}_{acc}(tx')$ or $\text{end}_{acc}(tx) = \text{end}_{acc}(tx')$, tx is a credit transaction and tx' is a debit transaction;

Let H be the permutation of $\mathcal{T}(\mathcal{E}, acc)$ implied by the total order "$<$". It is convenient to think of H as a sequence of epoch sets, i.e., $H = (E(0), E(1), \ldots, E(n), \ldots)$ or a sequence of groups, i.e., $H = (G_1(0), \ldots, G_1(n), G_2(n), G_3(n), \ldots)$.

Now, we need to prove that H is both legal and consistent with real-time partial order $\prec_{\mathcal{E},acc}$. We are doing this by induction on the length of the permutation H. We start with the base of the induction.

Lemma 2.10. *H up to $E(0)$ is consistent with $\prec_{\mathcal{E},acc}$ and legal.*

Proof. This follows from the fact that $E(0)$ contains only genesis transaction $tx_{init,\ acc}$ that deposits initial balance to the account, which by definition is non-negative. □

Now, assuming that H is legal and consistent with $\prec_{\mathcal{E},acc}$ up to $E(k)$, let us prove that it is legal and consistent with $\prec_{\mathcal{E},acc}$ up to $E(k+1)$.

First, we show that H is consistent with $\prec_{\mathcal{E},acc}$.

Lemma 2.11. *For any $i \in \{1, 2, 3\}$, for any $tx, tx' \in G_i(k+1)$ if $tx \prec_{\mathcal{E},acc} tx'$ then $tx < tx'$.*

Proof. Given $\text{end}_{acc}(tx) < \text{start}_{acc}(tx')$, we need to show that $\text{end}_{acc}(tx) < \text{end}_{acc}(tx')$. This is obvious as $\text{end}_{acc}(tx) < \text{start}_{acc}(tx') \leq \text{end}_{acc}(tx')$. □

Lemma 2.12. *For any $tx \in G_1(k+1)$ and $tx' \in G_2(k+1) \cup G_3(k+1)$: $tx' \not\prec_{\mathcal{E},acc} tx$.*

Proof. We prove this lemma by contradiction. Let us assume that tx' precedes tx. Consider two scenarios:

- tx is a debit transaction. Then, tx could not be accepted by $\text{COD}[acc][k+1]$ as it was closed before tx' was submitted to the $\text{CONSENSUS}[acc][k+2]$ according to the implementation. Contradiction.
- tx is a credit transaction. According to COD-Close Safety and algorithm implementation, as tx returned from $\text{COD}[acc][k+1]$, then it is both consensus input and output. This implies that it was committed by the time tx' was submitted to the $\text{CONSENSUS}[acc][k+2]$. Contradiction.

We came to a contradiction in both cases, thus for any $tx \in G_1(k+1)$ and $tx' \in G_2(k+1) \cup G_3(k+1)$ $tx' \not\prec_{\mathcal{E},acc} tx$. □

Lemma 2.13. *For any $tx \in G_2(k+1)$ and $tx' \in G_3(k+1)$: $tx' \not\prec_{\mathcal{E},acc} tx$.*

Proof. It follows from the fact that any pair of transactions $tx \in G_2(k+1)$ and $tx' \in G_3(k+1)$ should have been submitted as a part of $\text{CONSENSUS}[acc][k+2]$ input and both are part of its output, which implies that there should exist a time t when both tx and tx' are active. □

Lemma 2.14. *For any transaction $tx \in \bigcup_{i=0}^{k} E(i)$, and $tx' \in E(k+1)$: $tx' \not\prec_{\mathcal{E},acc} tx$.*

Proof. Let epochStart(n) be the time when a process receives a value from CONSENSUS[acc][n] for the first time. Note that epochStart(n) $\leq$ epochStart($n+1$).

By COD-Close Safety property of COD, and C-Validity of Consensus, for any $i \leq k$, the output of CONSENSUS[acc][$i+1$] contains (in $allCredits^\sigma$, $selectedDebits$, or $cancelledDebits$) all transactions from $E(i)$. Hence, $\forall tx \in E(i) : \text{start}_{acc}(tx) \leq \text{epochStart}(i+1)$. Moreover, by definition of $E(k+1)$, for any $tx' \in E(k+1)$, $\text{end}_{acc}(tx') \geq \text{epochStart}(k+1)$. Hence $\text{start}_{acc}(tx) \leq \text{epochStart}(i+1) \leq \text{epochStart}(k+1) \leq \text{end}_{acc}(tx')$.

□

Lemma 2.15. *If H is consistent with $\prec_{\mathcal{E},acc}$ up to $E(k)$, then it is consistent with $\prec_{\mathcal{E},acc}$ up to $E(k+1)$.*

Proof. From Lemma 2.11, we know that transactions inside every group for epoch $k+1$ are ordered such that if $tx \prec_{\mathcal{E},acc} tx'$, then $tx < tx'$.

Also, according to Lemma 2.12 and Lemma 2.13, ordering of groups is consistent as well: i.e., $\forall tx_1 \in G_1(k+1), tx_2 \in G_2(k+1), tx_3 \in G_3(k+1)$: $tx_1 < tx_2 < tx_3$ and it cannot be that a transaction from a higher group precedes a transaction from a lower group in the real-time order $\prec_{\mathcal{E},acc}$.

Finally, according to Lemma 2.14, ordering of transactions between epochs is consistent with the real-time order.

Taking all these facts into consideration together with the fact that H is consistent with $\prec_{\mathcal{E},acc}$ up to $E(k)$, we conclude that H is consistent with $\prec_{\mathcal{E},acc}$ up to $E(k+1)$.

□

Now, let us show that H is also legal.

Lemma 2.16. *Consider a debit transaction tx such that $tx \in G_1(k+1)$. Consider the first invocation of COD[acc][$k+1$].Submit that returns $\langle debits^\sigma, credits^\sigma \rangle$ such that $\langle tx, * \rangle \in debits^\sigma$. Then, for any credit transaction $tx_c \in credits^\sigma$: $tx_c < tx$.*

Proof. Note that by definition of $G_1(k+1)$, either $tx_c \in G_1(k+1)$ or $tx_c \in \text{Txs}(\text{COD}[acc][k+1].initCredits^\sigma)$. In the latter case, by Theorem 2.8, $tx_c \in E(n')$ for some $n' < n$ and, hence, $tx_c < tx$.

Consider the former case ($tx_c \in G_1(k+1)$). We need to prove that $\text{end}_{acc}(tx_c) < \text{end}_{acc}(tx)$. Let t be the moment when the invocation returned. It is easy to see that $t < \text{end}_{acc}(tx)$. Moreover, for any $tx_c \in credits^\sigma$: $\text{end}_{acc}(tx_c) = \text{commitTime}(tx_c) < t$ since $credits^\sigma$ includes a valid commit certificate for tx_c. Hence, $\text{end}_{acc}(tx_c) < t < \text{end}_{acc}(tx)$.

□

Recall that $\rho_{\mathcal{E},acc}(tx)$ maps the debit transaction tx to the return value of the corresponding Transfer operation in $\mathcal{E}$.

Also, recall that, for a debit transaction tx on acc and a permutation H of $\mathcal{T}(\mathcal{E}, acc)$, $S(H, tx)$ denotes the set of credit and successful debit transactions in the prefix of H up to, but not including, tx, i.e., $S(H, tx) = \{tx' \in \mathcal{T}(\mathcal{E}, acc) \mid tx' < tx$ and either $tx'.recipient = acc$ or $tx'.sender = acc$ and $\rho_{\mathcal{E},acc}(tx') = \text{OK}\}$.

Lemma 2.17. *For any debit $tx_f \in E(k+1)$ such that $\rho_{\mathcal{E},acc}(tx_f) = FAIL$, $balance(S(H, tx_f), acc) < tx_f.amount$.*

Proof. Note that $tx_f \in G_3(k+1)$. Also, $\forall tx \in S(H, tx_f)$: either $tx \in \bigcup_{i=0}^{k} E(i)$ or $tx \in G_1(k+1) \cup G_2(k+1)$.

Now, consider the first moment of time when tx_f was returned as a part of a consensus output on line 47 to some correct process, i.e., an invocation of CONSENSUS[acc][$k+2$] returned $\langle \langle credits^\sigma, selectedDebits, cancelledDebits \rangle, * \rangle$ such that $tx_f \in cancelledDebits$.

Consider any $tx \in S(H, tx_f)$. We want to prove that $tx \in \text{Txs}(credits^\sigma) \cup selectedDebits$. Indeed, consider two cases:

- $tx \in \bigcup_{i=0}^{k} E(i)$: by Theorem 2.8, any such transaction tx should be either present in COD[acc][$k+1$].$initDebits$ or COD[acc][$k+1$].$initCredits^\sigma$, and by COD-Close Safety $initCredits^\sigma \subseteq credits^\sigma$ and $initDebits \subseteq selectedDebits$.
- $tx \in G_1(k+1) \cup G_2(k+1)$: $\text{Txs}(credits^\sigma) \cup selectedDebits$ by the definition of groups $G_1(k+1)$ and $G_2(k+1)$ and COD-Close Safety property.

By Theorem 2.8 and the definition of $G_2(k+1)$, $\forall tx \in \mathit{selectedDebits} \cup \text{Txs}(\mathit{credits}^\sigma)$: $tx \in G_2(k+1), G_1(k+1)$, or $\left(\bigcup_{i=0}^{k} E(i)\right)$. Hence, $tx \in S(H, tx_f)$. Thus, $S(H, tx_f) = \mathit{selectedDebits} \cup \text{Txs}(\mathit{credits}^\sigma)$.

Now, recall that, according to the COD-Close Safety property, the resulted set *selectedDebits* should be "maximal by inclusion", i.e., $\nexists tx \in \mathit{cancelledDebits}$, such that TotalValue(Txs($\mathit{credits}^\sigma$)) $\geq$ TotalValue($\mathit{selectedDebits} \cup \{tx\}$).

Let us summarize all of the above:

1. $tx_f \in G_3(k+1) \Leftrightarrow tx_f \in \mathit{cancelledDebits}$;
2. $\nexists tx \in \mathit{cancelledDebits}$, such that TotalValue(Txs($\mathit{credits}^\sigma$)) $\geq$ TotalValue($\mathit{selectedDebits} \cup \{tx\}$);
3. $S(H, tx_f) = \mathit{selectedDebits} \cup \text{Txs}(\mathit{credits}^\sigma)$.

Hence, balance$(S(H, tx_f), acc) < tx_f.\mathit{amount}$. □

Lemma 2.18. *For any debit $tx_s \in E(k+1)$ such that $\rho_{\mathcal{E},acc}(tx_s) = OK$, balance$(S(H, tx_s), acc) \geq tx_s.\mathit{amount}$.*

Proof. Consider a debit transaction $tx_s \in E(k+1)$ such that $\rho_{\mathcal{E},acc}(tx_s) = \text{OK}$. Note that tx_s cannot belong to $G_3(k+1)$ as this group only contains failed transactions. Hence, tx_s belongs to either $G_1(k+1)$ or $G_2(k=1)$.

Suppose $tx_s \in G_1(k+1)$. Let $t = \text{end}_{acc}(tx_s)$. Let $\{o_i\}_{i=1}^{k}$ be the set of all COD$[acc][k+1]$.Submit operations such that o_i returned OK$(\mathit{debits}_i^\sigma, \mathit{outCredits}_i^\sigma)$ by the moment t. By COD-Submit Safety, TotalValue$(\mathit{initDebits} \cup \text{Txs}(\bigcup_{i=1}^{k} \mathit{debits}_i^\sigma)) \leq$ TotalValue$(\text{Txs}(\mathit{initCredits}^\sigma) \cup \text{Txs}(\bigcup_{i=1}^{k} \mathit{outCredits}_i^\sigma))$. Moreover, by Theorem 2.8 and the definition of $G_1(k+1)$, one can see that debits$(S(H, tx_s), acc) \cup \{tx_s\} \subseteq \mathit{initDebits} \cup \text{Txs}(\bigcup_{i=1}^{k} \mathit{debits}_i^\sigma))$ and debits$(S(H, tx_s), acc) \supseteq \text{Txs}(\mathit{initCredits}^\sigma) \cup \text{Txs}(\bigcup_{i=1}^{k} \mathit{outCredits}_i^\sigma)$. Hence, TotalValue(debits$(S(H, tx_s), acc) \cup \{tx_s\}) \leq$ TotalValue(debits$(S(H, tx_s), acc)$), which is equivalent to saying that balance$(S(H, tx_s), acc) \geq tx_s.\mathit{amount}$.

Now consider the case when $tx_s \in G_2(k+1)$. Let us have a look at *selectedDebits* and $\mathit{allCredits}^\sigma$ that are part of the consensus output (i.e., returned from Consensus$[acc][k+2]$.Propose). From COD-Submit Safety and the implementation of the algorithm, we know that TotalValue($\mathit{selectedDebits}$) $\leq$ TotalValue(Txs($\mathit{allCredits}^\sigma$)). Moreover, by Theorem 2.8 and the definition of $G_2(k+1)$, $\mathit{debits}(S(H, tx_s), acc) \cup \{tx_s\} \subseteq \mathit{selectedDebits}$ and $\mathit{credits}(S(H, tx_s)) = \text{Txs}(\mathit{allCredits}^\sigma)$. Hence, TotalValue(debits$(S(H, tx_s), acc) \cup \{tx_s\}) \leq$ TotalValue(debits$(S(H, tx_s), acc)$), which is equivalent to saying that balance$(S(H, tx_s), acc) \geq tx_s.\mathit{amount}$. □

Lemma 2.19. *If H is legal up to $E(k)$, then it is legal up to $E(k+1)$.*

Proof. This follows directly from Theorems 2.17 and 2.18.

□

Theorem 2.20. *CryptoConcurrency satisfies the Transfer Consistency property of Asset Transfer.*

Proof. This fact follows from Lemma 2.10, Lemma 2.15 and Lemma 2.19. □

Account Transactions Completeness. Let us now show that the implementation of the GetAccountTransactions operation correct, i.e., CryptoConcurrency satisfies the Account Transactions Completeness property.

Theorem 2.21. *CryptoConcurrency satisfies the Account Transactions Completeness property of Asset Transfer.*

Proof. The proof of this theorem follows from the implementation of the algorithm and the AOS-Output Validity property of the Append-Only Storage. Let us assume that operation GetAccountTransactions() is invoked by an owner of a correct account acc at time t_0 and returns set txs^σ.

Consider the first part of the property: (debits$(C(t_0), acc) \cup$ credits$(C(t_0), acc) \subseteq \text{Txs}(txs^\sigma)$. It directly follows from the definition of committed transactions, implementation of VerifyCommitCertificate function (via verifying function of Global Storage) and AOS-Consistency property of the Append-Only Storage (in particular, Global Storage).

Now, let us consider the second part of the property: $\forall \langle tx, \sigma \rangle \in txs^{\sigma}$: VerifyCommitCertificate$(tx, \sigma) = true$. This is follows from the implementation of GetAccountTransactions and VerifyCommitCertificate (both are implemented via Global Storage), and from the AOS-Output Validity property of the Append-Only Storage. □

Transfer Concurrency. We conclude the proof of correctness of CryptoConcurrency by showing that it satisfies the Transfer Concurrency property. Basically, this property states that if there are no overspending attempts on a correct account *acc* after some time t_0, then after some time $t_1 \geq t_0$, the owners of *acc* do not use consensus. Since we only use consensus to perform the Recovery procedure (line 47), the proof boils down to an argument that the number of the epochs is finite if the condition in the Transfer Concurrency property holds.

Recall that $C(t)$ denotes the set of all transactions committed by time t and $O(t, acc)$ denotes the set of all active debit transaction of *acc* at time t, i.e., $\{tx \mid tx.sender = acc$ and $\text{start}_{acc}(tx) \leq t \leq \text{end}_{acc}(tx)\}$. In the proofs, we consider a correct account *acc* and a time t_0 such that for all $t \geq t_0$: balance$(C(t), acc) \geq$ TotalValue$(O(t, acc) \setminus C(t))$.

We say that epoch e *starts* at the moment when the first correct replica initializes COD$[acc][e]$ (line 69) and *ends* when the next epoch $(e + 1)$ starts. We call a debit transaction *tx interfering* iff Transfer(tx) was invoked before t_0. Also, we say that debit transaction *tx* is *active in epoch e* iff it was active at some time between the start and the end of epoch e.

Lemma 2.22. *There exists an epoch e, such that there are no interfering transactions active in epoch e, or the number of the epochs is finite.*

Proof. This lemma follows from the Transfer Liveness property, which says that every Transfer operation eventually terminates, and the fact that there can only be a finite number of interfering transactions. □

Lemma 2.23. *The number of epochs is finite.*

Proof. We prove this lemma by contradiction. Let us assume that the number of the epochs is infinite. Then, by Lemma 2.22, there should exist an epoch e^*, such that there are no interfering transactions active in epoch e^*. Note that e^* must have started after t_0 (by definition of an interfering transaction).

As number of the epochs is infinite, there should exist an epoch $e^* + 1$. We know from the implementation that correct account can progress into an epoch $e^* + 1$ only in case one of the clients returned FAIL from COD$[acc][e^*]$.Submit$(\ldots)$. According to the COD-Submit Success of COD object, it can only happen if there exists a set $S = \{op_1, \ldots, op_n\}$, where $op_i =$ COD$[acc][e^*]$.Submit$(debits_i, credits_i^{\sigma}))$, such that TotalValue$(debits) >$ TotalValue$(credits)$, where $debits = \bigcup_{i \in S} debits_i \cup initDebits$ and $credits = \text{Txs}(\bigcup_{i \in S} credits_i^{\sigma} \cup initCredits^{\sigma})$.

Let us consider a minimal (by inclusion) such set S. Each Submit operation can be naturally associated with a Transfer operation by which it was invoked. Let us sort the Submit operations in S by the beginning time of the associated Transfer operations in ascending order. We consider the "largest" (w.r.t. the above sorting) operation $op_{max} =$ COD$[acc][e^*]$.Submit$(debits_{max}, credits_{max}^{\sigma})$. Note that $credits_{max}^{\sigma}$ are committed credits and were read from the GlobalStorage. From the condition imposed by Transfer Concurrency, we also know that for all $t \geq t_0$, balance$(C(t), acc) \geq$ TotalValue$(O(t, acc) \setminus C(t))$. Combining these facts together, we can easily see that TotalValue$(\text{Txs}(initCredits^{\sigma} \cup credits_{max}^{\sigma})) >$ TotalValue$(\bigcup_{i \in S} debits_i \cup initDebits)$. Then such set S does not exist. A contradiction.

□

Theorem 2.24. *CryptoConcurrency satisfies the Transfer Concurrency property of Asset Transfer.*

Proof. In Lemma 2.23, for a correction account *acc*, we showed that the number of the epochs is finite assuming that there exist t_0, such that for all $t \geq t_0$: balance$(C(t), acc) \geq$ TotalValue$(O(t, acc) \setminus C(t))$. According to the implementation, if number of the epochs is finite, then from some moment of time t_1, no consensus objects are invoked after time t_1 on *acc*. Thus, CryptoConcurrency satisfies the Transfer Concurrency property of Asset Transfer.

□

2.12.2 Proof of Correctness: Closable Overspending Detector

In this subsection, we demonstrate the correctness of our implementation of the Closable Overspending Detector abstraction, the most important building block of CryptoConcurrency, as specified in Algorithms 2.5 to 2.8.

COD-Submit Validity. The next two lemmas will help us to prove the COD-Submit Validity property of COD.

Lemma 2.25. *If a correct client returns OK($debits^\sigma$, $outCredits^\sigma$) from Submit($debits$, $credits^\sigma$), then $debits = Txs(debits^\sigma)$.*

Proof. This lemma directly follows from the implementation. When returning from the Accept function, the client forms $debits^\sigma$ from $debits$ transactions passed to the function (line 151). Note that a correct client passes $debits$ transactions from the Submit($debits$, $credits^\sigma$) to the Accept function. □

Lemma 2.26. *If a correct client returns OK($debits^\sigma$, $outCredits^\sigma$) from Submit(...), then $\forall \langle tx, \sigma_{accept} \rangle \in debits^\sigma$: $VerifyCODCert(tx, \sigma_{accept}) = true$.*

Proof. This lemma directly follows from the implementation. A correct client forms a certificate for each transactions that will be accepted by the verifying function VerifyCODCert. □

Lemma 2.27. *If a correct client returns OK($debits^\sigma$, $outCredits^\sigma$) from Submit(...), then $\forall \langle tx, \sigma_{accept} \rangle \in outCredits^\sigma$: $VerifyCODCert(tx, \sigma_{accept}) = true$.*

Proof. This lemma directly follows from the implementation. A correct client uses only committed credit transactions. □

Theorem 2.28. *Our implementation of Closable Overspending Detector satisfies the COD-Submit Validity property.*

Proof. This theorem follows directly from Lemma 2.25, Lemma 2.26 and Lemma 2.27. □

COD-Submit Safety. The next property of COD that we will address is the COD-Submit Safety.

A certificate $\sigma_{prepare}$ is called a *prepare certificate* for a set $debits$ iff it is a threshold signature of a message $\langle$PrepareResp, $debits\rangle$ (formed at line 142). If a set of debit transactions $debits$ has a prepare certificate, then it is called *prepared*.

Let us consider all prepared sets of debits transactions for a COD object.

Lemma 2.29. *Any two prepared debit sets $debits_i$ and $debits_j$ are comparable (w.r.t. $\subseteq$): either $debits_i \subseteq debits_j$ or $debits_j \subset debits_i$.*

Proof. Let us consider a prepared set of debits $debits_i$. As it is prepared, there exists a prepare certificate $\sigma_{prepare_i}$. As a prepare certificate is, essentially, a threshold signature, then there exists a quorum Q that signed message $\langle$PrepareResp, $debits_i\rangle$.

Similarly, there exists a quorum Q' that signed a message $\langle$PrepareResp, $debits_j\rangle$ for $debits_j$.

Due to the quorum intersection property there exists a correct replica r, such that $r \in Q \cap Q'$. This implies that r signed both $\langle$PrepareResp, $debits_i\rangle$ and $\langle$PrepareResp, $debits_j\rangle$. Note that correct replicas sign only comparable sets of transactions (w.r.t. $\subseteq$). Thus, either $debits_i \subseteq debits_j$ or $debits_j \subset debits_i$. □

Lemma 2.30. *The total amount spent by initDebits and transactions accepted by a COD instance never exceeds the sum of the committed credits for the account COD.acc.*

Proof. Let us note that $initDebits$ is covered by $initCredits^\sigma$ by definition. Also, from the implementation it follows that any prepared set of debits contains $initDebits$.

According to the implementation it holds that if tx is accepted, then there exists a prepared set of debit transactions $debits$, such that $tx \in debits$. By Lemma 2.29, all prepared set of debits are related by containment. Then, for any finite set S of accepted transactions, there exists a prepared set $debits_{all}$, such that it contains all transactions from S. A corresponding prepare certificate for $debits_{all}$ is a threshold signature formed from signatures of of $\langle$PrepareResp, $debits_{all}\rangle$ of some quorum Q. In

every quorum there are at least f signatures made by correct replicas. Note that correct replicas only sign $\langle \text{PREPARERESP}, debits_{all} \rangle$ if they saw enough committed credits (line 176). Consequently, the total amount spent by *initDebits* and transactions accepted by a COD instance never exceeds the sum of committed credits for the account COD.*acc*. □

Now, we want to prove that if *acc* is a correct account, then the total amount of initial debits and debits accepted by the COD by time t, does not exceed the total amount of all credits returned by the Submit operation by time t. We prove this by showing that our implementation satisfies even stronger property in the following lemma.

Lemma 2.31. *For any subset $S = \{o_1, \ldots, o_k\}$ of Submit operations invoked by correct clients such that $\forall i \in \{1, \ldots, k\}$: o_i returns $OK(debits_i^\sigma, outCredits_i^\sigma)$, it holds that $TotalValue(initDebits \cup \bigcup_{i=1}^{k} debits_i^\sigma) \leq TotalValue(initCredits^\sigma \cup \bigcup_{i=1}^{k} credits_i^\sigma)$.*

Proof. Let us consider a set of operations S from the lemma condition. From the COD-Submit Validity property, we know that if there exists $i \in \{1, \ldots, k\}$, such that $tx \in debits_i^\sigma$, then tx is accepted. Let us match any operation $o_i \in S$ with prepared set of debit transactions $preparedDebits_i$ that was collected during execution of a given operation. By Lemma 2.29, all prepared sets of debit transaction are comparable (w.r.t., $\subseteq$). Thus, in the set $\{preparedDebits_i\}_{i=1}^{i=k}$, there should exist maximum set $preparedDebits_m$.

A corresponding prepare certificate for $preparedDebits_m$ is a threshold signature formed from signatures of of $\langle \text{PREPARERESP}, preparedDebits_m \rangle$ of some quorum Q. In every quorum there are at least f signatures made by correct replicas. Note that correct replicas only sign $\langle \text{PREPARERESP}, preparedDebits_m \rangle$ if they accounted enough committed credits (line 176) to cover all transactions from there. As $outCredits_m^\sigma$ contains all credits correct replicas saw before signing the message, TotalValue(Txs($outCredits_i^\sigma$)) $\geq$ TotalValue($preparedDebits_m$). From the implementation, Txs($debits_m^\sigma$) $\subseteq$ $preparedDebits_m$. Even more, $\forall i \in \{1, \ldots, k\}$: Txs($debits_i^\sigma$) $\subseteq$ $preparedDebits_i$. Also, recall that $\forall i \in \{1, \ldots, k\}$: $preparedDebits_i \subseteq preparedDebits_m$. Hence, TotalValue($initDebits \cup \bigcup_{i=1}^{k} debits_i^\sigma$) $\leq$ TotalValue($initCredits^\sigma \cup \bigcup_{i=1}^{k} credits_i^\sigma$). □

Lemma 2.32. *If $tx \in restrictedDebits$, then tx is never accepted by a COD object.*

Proof. This lemma follows from the implementation. Correct replicas do not respond to messages containing transactions from *restrictedDebits* during Prepare phase (line 170). □

Theorem 2.33. *Our implementation of Closable Overspending Detector satisfies the COD-Submit Safety property.*

Proof. This theorem follows from Lemma 2.30, Lemma 2.31 and Lemma 2.32. □

COD-Submit Success. The next step is to show that our COD protocol satisfies the COD-Submit Success property.

Theorem 2.34. *Our implementation of Closable Overspending Detector satisfies the COD-Submit Success property.*

Proof. We prove this theorem by contradiction. Let us assume that *acc* is a correct account, no client invokes the Close operation, and for every possible subset S of invoked operations Submit($debits_i, credits_i^\sigma$): TotalValue($debits$) $\leq$ TotalValue($credits$) (where $debits = \bigcup_{i \in S} debits_i \cup initDebits$ and $credits$ = Txs($\bigcup_{i \in S} credits_i^\sigma \cup initCredits^\sigma$)); however there exists an operation Submit($debits_p, credits_p^\sigma$) invoked by a client p that returns FAIL. Before we continue with the proof, let us note that any valid reply $\langle \text{PREPARERESP}, credits^\sigma, debits^\sigma, sig \rangle$ from a replica should satisfy the following condition: $\forall \langle tx, \langle deps_{tx}, sig_{tx} \rangle \rangle \in debits^\sigma$, $\forall id \in deps_{tx}$: $\exists \langle tx', \sigma_{tx'} \rangle \in credits^\sigma$ such that $id = tx'.id$. Also, $\forall i$: $initDebits \subseteq$ Txs($debits_i^\sigma$) and $initCredits^\sigma \subseteq credits_i^\sigma$.

Now, consider operation op = Submit($debits_p, credits_p^\sigma$) executed by a correct client p. Note that op can only return FAIL from the Prepare function, as Accept returns FAIL only after receiving CLOSED reply, and this is impossible, since no client invokes the Close operation.

Thus, client p should have collected a set of messages $M = \{\langle \text{PREPARERESP}, debits_k^\sigma, credits_k^\sigma, sig_k \rangle\}_{k \in \{1, \ldots, |M|\}}$ from a quorum Q. Otherwise, either

Close operation was invoked by some client, which is impossible by Thus, FAIL is returned by the Prepare function. To be more precise it is returned at line 140. It means that, naturally, there should exist a set S' of Submit($debits_i, credits_i^\sigma$) operations and a set of credits $credits_{extra}^\sigma$, such that $\bigcup_{i \in S'} debits_i \cup initDebits = \text{Txs}(\bigcup_{k \in M} debits_k^\sigma)$ and $\bigcup_{i \in S'} credits_i \cup initCredits^\sigma = \text{Txs}(\bigcup_{k \in M} credits_k^\sigma \cup credits_{extra}^\sigma)$. As we assume that p returns FAIL, then $\text{TotalValue}(\text{Txs}(\bigcup_{k \in M} debits_k^\sigma)) > \text{TotalValue}(\text{Txs}(\bigcup_{k \in M} credits_k^\sigma \cup credits_{extra}^\sigma))$. Then, it also means that $\text{TotalValue}(\bigcup_{i \in S'} debits_i \cup initDebits) > \text{TotalValue}(\bigcup_{i \in S'} credits_i \cup initCredits^\sigma)$. However, this contradicts with the assumption that for any subset S' of invoked Submit($debits_i, credits_i^\sigma$), such that $\text{TotalValue}(debits) > \text{TotalValue}(credits)$ (where $debits = \bigcup_{i \in S'} debits_i \cup initDebits$ and $credits = \text{Txs}(\bigcup_{i \in S'} credits_i^\sigma \cup initCredits^\sigma)$). Consequently, our assumption that there exists a Submit($debits_p, credits_p^\sigma$) operation, which returns FAIL, is wrong. This means that our implementation of COD satisfies COD-Submit Success property. □

COD-Close Validity. Now we continue with the proof of the COD-Close Validity property of the COD object, which is relatively simple.

Theorem 2.35. *Our implementation of Closable Overspending Detector satisfies the COD-Close Validity property.*

Proof. Trivially follows from the algorithm implementation. □

COD-Close Safety.

Lemma 2.36. *If a correct client returns $\langle CODState, \sigma_{state} \rangle$ from Close(pendingDebits), then:*

- $\forall tx \in pendingDebits : tx \in CODState.selectedDebits$ *or* $tx \in CODState.cancelledDebits$;
- $initDebits \subseteq CODState.selectedDebits$ *and* $restrictedDebits \subseteq CODState.cancelledDebits$;
- $CODState.selectedDebits \cap CODState.cancelledDebits = \emptyset$;
- $TotalValue(Txs(CODState.credits^\sigma)) \geq TotalValue(CODState.selectedDebits)$

Proof. This lemma trivially follows from the implementation of the SplitDebits function (lines 112 to 122) and the way it is used in the implementation of the Close operation (lines 153 to 162). □

Lemma 2.37. *If a correct client p returns $\langle CODState, \sigma_{state} \rangle$ from Close(pendingDebits), then for every transaction tx accepted by this COD: $tx \in CODState.selectedDebits$.*

Proof. Let us consider an accepted transaction tx. If tx is accepted, then there exists a certificate σ_{accept}, such that $\text{VerifyCODCert}(tx, \sigma_{accept}) = true$. This means that σ_{accept} contains a threshold signature formed by a client q from valid signatures $\{sig_r\}_{r \in Q}$ made by a quorum of replicas Q.

As p is a correct client, then during Close operation it should have collected valid CloseResp responses from a quorum Q'. By quorum intersection property there exists a correct replica r, such that $r \in Q \cap Q'$. Then, one of the following should have happened before the other: (i) r produced a signature sig_r and sent $\langle \text{AcceptAck}, sig_r \rangle$ to q; (ii) r sent to p a valid CloseResp reply. Note that as r is correct, (ii) could not happen before (i). In this case, r would send Closed message to q. Thus, (i) happened before (ii). As r is correct, it signed *preparedDebits* (such that $tx \in preparedDebits$) and attached it to CloseResp that was sent to p. As p is correct and due to the way SplitDebits is implemented: $preparedDebits \subseteq CODState.selectedDebits$.

The above reasoning is valid for any accepted transaction tx and any Close operation performed by a correct client. □

Lemma 2.38. *If a correct client obtains $\langle CODState, \sigma_{state} \rangle$ from Close(pendingDebits), then for any Submit operation that returns $OK(debits^\sigma, outCredits^\sigma)$ to a correct client: $outCredits^\sigma \subseteq CODState.credits^\sigma$ and $\forall \langle tx, \sigma \rangle \in CODState.credits^\sigma$: $VerifyCommitCertificate(tx, \sigma) = true$. Moreover, $initCredits^\sigma \subseteq CODState.credits^\sigma$.*

Proof. The first part of the lemma follows from the implementation of Accept phase. Upon receiving AcceptRequest message, correct replicas add $outCredits^\sigma$ to their local set of credit transactions (line 185). For any such set there exist at least $f+1$ correct replicas that store these credit transactions. Then, by quorum intersection property, during any execution of Close operation invoked by client c, $\forall tx \in outCredits^\sigma$ will be read by c.

The second part of the lemma follows from the fact that replicas ignore messages if credits do not come together with commit certificates (line 184).

□

Theorem 2.39. *Our implementation of Closable Overspending Detector satisfies the COD-Close Safety property.*

Proof. This theorem follows from Lemma 2.36, Lemma 2.37 and Lemma 2.38. □

COD-Liveness. Finally, we want to show that any operation invoked by a correct process eventually returns and, hence, to prove that our implementation satisfies the COD-Liveness property.

Let us recall that once a correct client returns FAIL from an invocation of Submit, it never invokes this operation on a given COD object again.

Lemma 2.40. *Every Close operation invoked by a correct client eventually returns.*

Proof. The lemma follows from the fact that the operation makes a constant number of steps, all **wait for** conditions will be satisfied as the client waits for a quorum of replies and there exists a quorum that consists of only correct replicas that eventually responds. □

Lemma 2.41. *If a correct client p does not return from Submit operation, then Close is never invoked.*

Proof. Indeed, otherwise, if Close is invoked, then p will eventually receive Closed reply and return FAIL □

Lemma 2.42. *If some correct client invokes Submit, then some (not necessarily the same) correct client eventually returns from Prepare function.*

Proof. We prove this lemma by contradiction. Let us assume that all processes that invoked Submit operation of a given *COD* object never return from Prepare function. Note, that in this case Close is never invoked (Lemma 2.41). As an account is shared by a finite number of clients, there is a finite number of Submit($debits_i, credits_i^\sigma$) invocations. Due to the fact that the number of invocations is finite, eventually every client will receive a set of equal $\langle$PrepareResp, $debits^\sigma, credits^\sigma\rangle$ replies from some quorum Q. No client can return FAIL (by the assumption), however as a client obtains a set of equal replies, the condition at line 141 will be satisfied and it will return OK(...). This contradicts with our initial assumption. Thus, if some correct client invokes Submit, then some correct client eventually returns from Submit operation. □

Lemma 2.43. *If a correct client invokes the Accept function while executing the Submit operation, it eventually returns from it.*

Proof. The proof of this lemma directly follows from the implementation: i.e., it should eventually get a quorum of valid AcceptAck messages or at least one valid CloseResp message. □

Lemma 2.44. *If a correct client p invokes Submit($debits, credits^\sigma$), then it eventually returns from it.*

Proof. We prove this lemma by contradiction. Let us assume that p never returns from the Submit function. The only place it can stuck is while executing Prepare function: indeed, correct processes always return from Accept by Lemma 2.44.

Now, let us consider two scenarios: (i) there exists a time t_0, starting from which no client returns from the Submit operation, (ii) no such time t_0 exists, i.e., for any t, there always exists some client q, which returns from Submit function, after time t.

We start with the first scenario. As an account is shared by a finite number of clients, there is a finite number of Submit($debits_i, credits_i^\sigma$) invocations. Let us consider all operations Submit($debits_i, credits_i^\sigma$) that are active after time t_0. There should exist time $t_1 \geq t_0$, such that all operations that do not return

from Prepare are active at all times $t > t_1$. There exists at least one operation that is active at that time – one that was invoked by client p. Due to the fact that the number of invocations is finite, eventually every client will receive a set of equal $\langle \text{PREPARERESP}, debits_i^\sigma, credits_i^\sigma, sig_i \rangle$ replies from some quorum Q. No client can return FAIL (by the assumption that clients do not exit Prepare function), however as a client obtains a set of equal replies, the condition at line 141 will be satisfied and it will return OK(...) from the Prepare function. This way, client p returns from the Submit operation. This contradicts our assumption, and, consequently, (i) is impossible.

Let us consider now the second scenario: for any t, there always exists some client q, which returns from Submit operation, after time t. Note that the first $\langle \text{PREPARE}, debits^\sigma, credits^\sigma, submitDebits^\sigma \rangle$ message client p sent during the execution of Submit operation will eventually reach all correct replicas and they will add all transactions from $submitDebits^\sigma$ to their local set of debit transactions (line 175). Let denote the time it happens as t_1. From the scenario we consider, there should be an infinite number of Submit invocations on a given COD object: indeed, otherwise there should exists a time t_n, such that no operation returns after t_n. As a consequence, there should exist a Submit operation, which was invoked after t_1 and returned OK(...) (recall that correct client do not invoke Submit once it returns FAIL). As it started after t_1, client should have collected a set S of $\langle \text{PREPARERESP}, debits_i^\sigma, credits_i^\sigma, sig_i \rangle$, such that (i) $\forall i$: $submitDebits^\sigma \subseteq debits_i^\sigma$ (ii) every message in S has equal set of $debits_i^\sigma$. Then, there exists a prepare set of debits that includes all of the transactions from $submitDebits^\sigma$. Hence, eventually client p should return from the Prepare function at line 137, and then complete Submit operation after returning from the Accept function. However, this contradicts with our assumption, this means that (ii) is impossible.

We considered two potential scenarios and came to a contradiction in both of them. Note that the set of scenarios is exhaustive. Hence, □

Theorem 2.45. *Our implementation of Closable Overspending Detector satisfies the COD-Liveness property.*

Proof. This theorem follows from Lemma 2.40 and Lemma 2.44. □

Theorem 2.46. *Algorithm 2.7 and Algorithm 2.8 correctly implement Closable Overspending Detector.*

Proof. All properties of Closable Overspending Detector hold:

- COD-Submit Validity follows from Theorem 2.28;
- COD-Submit Safety follows from Theorem 2.33;
- COD-Submit Success follows from Theorem 2.34;
- COD-Close Validity follows from Theorem 2.35;
- COD-Close Safety follows from Theorem 2.39;
- COD-Liveness follows from Theorem 2.45.

□

2.12.3 Proof of Correctness: Append-Only Storage

In this subsection we show that our implementation of the Append-Only Storage is correct, i.e., it satisfies AOS-Consistency, AOS-Input Validity, AOS-Output Validity and AOS-Liveness.

Lemma 2.47. *For any value v and a key k, if there exists σ_{AOS} such that VerifyStoredCert$(k, v, \sigma_{AOS}) =$ true at the moment when ReadKey(k) was invoked by a correct client, then the output of this operation vs_{out}^σ will contain v (paired with a certificate), i.e., $\langle v, * \rangle \in vs_{out}^\sigma$.*

Proof. If there exists σ_{AOS} such that VerifyStoredCert$(k, v, \sigma_{AOS}) = true$, then there should exist a quorum of processes Q and vs^σ, such that every process $r \in Q$ signed a message $\langle \text{APPENDKEYRESP}, k, \text{MerkleTree}(\{v \mid \langle v, * \rangle \in vs^\sigma\}).root \rangle$. Note that any correct process $r \in Q$ should have also added vs^σ in their $log[k]$.

Let us consider a correct client p that returns vs^{σ}_{out} from ReadKey(k) invoked when there existed σ_{AOS} such that VerifyStoredCert(k, v, σ_{AOS}) = *true*. Then, there should exist a quorum of processes Q' that responded with $\langle \text{READKEYRESP}, vs^{\sigma}_{i} \rangle$ messages. According to the quorum intersection property, there should exist a correct replica r, such that $r \in Q \cap Q'$. As r added v (with its certificate) to its $log[k]$, it should have also included it into $\langle \text{READKEYRESP}, vs^{\sigma}_{r} \rangle$ that it replied with to p. By the implementation, $vs^{\sigma}_{out} = \bigcup_i vs^{\sigma}_{i}$. Thus p includes v (with its certificate) in vs^{σ}_{out}. □

Theorem 2.48. *Algorithm 2.9 is a correct implementation of the Append-Only Storage.*

Proof. All properties of the Append-Only Storage hold:

- AOS-Input Validity follows from the implementation;
- AOS-Output Validity follows from the implementation;
- AOS-Consistency follows from Lemma 2.47;
- AOS-Liveness follows from the fact that each operation makes a constant number of steps and that any **wait for** condition will be satisfied, as there exist $n - f$ correct processes that form a quorum and must eventually respond.

□

2.13 Latency Proofs

Lemma 2.49. *Latency of the AppendKey operation of Append-Only Storage invoked by a correct client is* 1 *round-trip.*

Proof. Follows directly from the implementation (Algorithm 2.9, lines 205 to 214). The client simply sends the request to the replicas and waits for signed acknowledgments from a quorum. □

Lemma 2.50. *Latency of the ReadKey operation of Append-Only Storage invoked by a correct client is* 2 *round-trips.*

Proof. Similarly, this fact follows directly from the implementation (Algorithm 2.9). In the algorithm, the client first waits for a quorum of READKEYRESP replies (1 round-trip) and then performs a write-back. The latency of the latter is 1 RTT as shown in Theorem 2.49. □

Theorem 2.51. *CryptoConcurrency exhibits k-overspending-free latency (as defined in Section 2.5) of $k + 4$ round-trips.*

Proof. In the definition of k-overspending-free latency, we only consider operations that start after the system stabilizes from all overspending attempts, i.e., when no epoch changes happen during the operation execution. In this case, the client will do the following sequence of actions:

- Execute ReadKey on lines 5, 7 and 13 and AppendKey on line 10 concurrently. By Theorems 2.49 and 2.50, this step will take at most 2 round-trips;
- Send the INITCOD message to replicas (line 17). This step does not affect the latency, as the client does not wait for replicas' replies and moves on directly to the next step;
- Execute COD[*acc*][*e*].Submit on line 19. As we discuss in the next paragraph, this step takes at most $k + 1$ round-trips;
- Finally, execute GLOBALSTORAGE[*acc*].AppendKey on line 23. By Theorem 2.49, this step will take 1 round-trip.

All we have left to show is that each client will return from the Submit operation of COD[*acc*][*e*] after at most $k + 1$ RTTs. We consider the Prepare phase first and show that the upper bound on the number of loop iterations (line 132) is k. The client c can return from the loop when it either (i) detects an overspending attempt (line 139) or (ii) converges on the debit sets received from a quorum of replicas

(line 141). Note that (i) is impossible given the conditions of this theorem, so we can safely consider only (ii).

For all debit transactions that terminated before the client started its Transfer operation, the client will observe them in the Account Storage and, thus, will include them in its input to COD. Hence, each time, after receiving a quorum of valid replies from the replicas, either the client learns about at least one new concurrent debit transaction, or terminates. Thus, after at most k round-trips, there will be no new transactions to learn and the client will exit the Prepare phase on line 143.

Finally, the latency of the Accept phase is always 1 round-trip, which results in a latency of $k + 1$ RTTs for COD[acc][e].Submit operation. □

Proof of Theorem 2.1 (Section 2.5). Follows from the theorems claiming that CryptoConcurrency satisfies all the properties of an asset transfer system with Transfer Concurrency (Theorems 2.4, 2.5, 2.7, 2.20, 2.21 and 2.24) as well as Theorem 2.51 asserting the k-overspending-free latency of CryptoConcurrency. □

2.14 Concluding Remarks

There are multiple interesting directions for future work.

First, our algorithm leaves space for further optimizations in addition to the ones we discussed in Section 2.8.4. To preclude the system state and protocol messages from growing without bound, one can introduce a checkpointing mechanism. For example, checkpointing can be implemented using occasional invocations of consensus similar to how our protocol resolves overspending attempts. Additionally, a practical implementation should probably avoid repeatedly exchanging (potentially, large) sets of transactions between the replicas and the clients and instead should send only the updates ("deltas") since their last communication.

Second, to quantify the actual performance gains of CryptoConcurrency, a practical implementation and a comparison to other similar protocols such as [118] would be of interest.

Third, recent results [76, 92, 93, 125] demonstrate how asynchronous Byzantine fault-tolerant systems can be *reconfigured* without relying on consensus. Notably, in [92], it is shown that a permissionless Proof-of-Stake asset transfer system can be implemented using a similar technique. A fascinating challenge is to combine reconfiguration with the ideas of CryptoConcurrency for building a *reconfigurable*, or even *permissionless*, asset transfer system with shared accounts and without a central consensus mechanism.

Finally, we believe that our work opens the way to *optimally-concurrent* solutions to other, more general problems, such as fungible *token smart contracts* [12], an abstraction intended to grasp the synchronization requirements of a specific type of Ethereum smart contracts. The objects allow the set of account owners to vary over time, which might require generalizing our notion of conflicts.

In the general case, it is appealing to address the question of optimally-concurrent state machine replication. Intuitively, one would like to avoid consensus-based synchronization whenever any reordering of concurrent operations has the same effect. Formalizing this intuition and implementing this kind of optimality in the context of generic state machine replication is left for future work.

Chapter 3

Linear View Change in Consensus with Optimistic Fast Track

3.1 Introduction

The ultimate vision for blockchain is to replace trusted parties by decentralized systems where collusion of a large fraction of the participants would be required for the attacker to break the system rules. To this end, *Byzantine consensus* [101] is typically used to reach agreement on a sequence of blocks of operations.

Intuitively, in a single instance of consensus, each participant, which we dub as *party*, starts with some valid value (e.g., a block of transactions or an availability certificate as in [50]). The goal is for each party to irrevocably output a valid value, which we dub as *decision*. Moreover, all parties that follow the protocol must output the same value. This property is known as *consistency*. Some parties are denoted as "corrupt" or "Byzantine", which means that they are controlled by a malicious adversary and may arbitrarily deviate from the protocol. Non-corrupt parties are denoted as "honest".

In this work, we are concerned with a much studied class of protocols designed to operate under a network model known as *partial synchrony* [60]. In this model, there is an unknown moment in time, called the *global stabilization time (GST)*, such that any message sent after this moment is delivered within a known time bound Δ. In order to ensure liveness of the protocols, this bound has to be pessimistic and, in practice, most messages are delivered much faster. Hence, we use δ to denote the unknown actual message delay in a given execution ($\delta < \Delta$). In this model, consensus is solvable only when the adversary corrupts less than one-third of the parties [60].

3.1.1 Fast track and linear communication

In order for blockchain systems to be competitive with centralized services, it is imperative to keep the latency of the consensus algorithm as low as possible while maintaining high throughput. In this work we are concerned with the subclass of Byzantine consensus protocols, known as *leader-based*, which achieve the best known parameters, as soon as some conditions, which we denote as *normal*, are met. These protocols require for their operation a mechanism that notifies the parties of the identity of one of them, denoted as *leader*, which may change in time. Broadly speaking, a time frame in which all players are aware of the same leader, is often designated as a *view*. A view is denoted as *normal* if the leader is honest and GST happened before the start of the view. Leader-based consensus protocols guarantee that, in a normal view, a common decision is reached within a fixed number of message delays. Moreover, architectures based on frequent changes of leaders [142] enable some form of *fairness*, i.e., that every player, when promoted as leader in a normal view, can enforce a common decision on its own input.

PBFT [40] is by far the most famous leader-based Byzantine consensus protocol. However, its quadratic communication complexity in the number of parties prohibits having a large number of them, which undermines the potential for decentralization. Hence, protocols with worst-case linear communication in every view, such as [1, 142], were proposed. They were designed to replace PBFT in blockchain applications. In this work, we will follow the baseline of the protocol Hot-Stuff v1 [1]. It credits its view-change mechanism to Tendermint [32], thus we will dub it as TH1. If the first view is normal,

protocol	first view			view change (worst case)	
	auth. complexity	latency		auth. complexity	latency
		fast track	slow track		
PBFT [40]	$O(n^2)$	—	3δ	$O(n^2)$	1δ
FaB Paxos [109]	$O(n^2)$	2δ	3δ	$O(n^2)$	1δ
SBFT [74]	$O(n)$	2δ	4δ	$O(n^2)$	1δ
TH1 [1, 32]	$O(n)$	—	4δ	$O(n)$	1Δ
HotStuff [142]	$O(n)$	—	6δ	$O(n)$	1δ
PnS [8]	$O(n)$	—	4δ	$O(n\phi)$ (1)	1δ
Wendy [73]	$O(n)$	—	4δ	$O(n)$	3δ (2)
this work	$O(n)$	2δ	4δ	$O(n)$	1Δ

1. ϕ is the number of views before the network stabilizes and an honest leader is elected. In practice, this number is usually small, but it can be arbitrarily large in theory.

2. In the design of Wendy, an extra round-trip was deliberately added to the worst case latency of the view change in order to save some computation and communication in the good case.

Table 3.1: Overview of existing and proposed protocols.

then a decision is reached within the delay of 4 consecutive message deliveries at the actual speed of the network, i.e., 4δ. This is one more message delay compared to PBFT (which needs 3 message delays to reach a decision). Then in every subsequent view, the new leader (dubbed as "proposer" in [1]) initially waits for the pessimistic bound on the message delay, Δ, before it starts proposing a new value. Notice that this waiting for a fixed delay is removed in subsequent works, either at the cost of more consecutive interactions [130, 142], or more cryptographic computations [7, 8, 73]. In this chapter, we only focus on the original approach of [1] and defer the integration with other protocols to a follow-up.

Meanwhile, Martin and Alvisi [109] demonstrated that PBFT itself is not optimal in terms of latency. More precisely, it is possible to reach a decision in just 2 message delays under some optimistic assumptions. This desirable guarantee is known as a *fast track*. Furthermore, if these assumptions do not hold, the protocol still reaches a decision in 3 message delays, on par with PBFT. A number of practical systems [74, 90] have further improved and implemented in practice the idea of a fast track. Unfortunately, these systems inherit the quadratic communication complexity of PBFT.

3.1.2 Our contributions

In this chapter, we suggest a protocol that combines the best of the two worlds: linear authenticator complexity and a fast track. If the network is synchronous from the beginning of the execution and all parties are honest, the proposed protocol reaches a decision in just 2 message delays. Otherwise, it falls back to TH1.

Interactive proofs that a value is safe. Our key technical contribution is the construction of an efficient *Proof of Exclusivity* (or PoE for short) sub-protocol. Intuitively, it allows the leader to prove to parties that no value other the one being proposed could have been decided on the fast track. It allows to safely integrate the fast track with the slow track without extra costs in communication complexity.

Structural changes to BFT. To preserve the latency of the BFT, despite the interactions needed in the PoE sub-protocol, we make a structural change to previous BFTs with a fast track. Namely, we allow players to cast preliminary votes for a value before they were convinced that this value is safe.

Optimistic zero-overhead track and accountability. Finally, in Section 3.7, we introduce the notion of an *optimistic zero-overhead PoE*. By reusing the signatures produced in the baseline protocol itself, we obtain a PoE at no extra cost unless the leader of the first view is caught on producing 2 contradictory digital signatures (the behavior we call "equivocation"), in which case the protocol still produces a PoE, but using up to 5 extra threshold signatures. If combined with a reconfiguration [100] or a slashing [33] mechanism, it is possible to make sure that, in the vast majority of cases, the protocol incurs

almost no extra communication or computational overhead compared to the baseline while providing 2 times smaller latency in the good case.

A detailed comparison of our protocol with previous works is shown in Table 3.1. There, authenticator complexity (shortened to "auth. complexity") is the total number of authenticators sent over the network, where an authenticator is either a message authentication code (implicitly appended to each message), a signature, or a threshold signature. The latency is measured since the start of the view and until at least one honest party reaches a decision (assuming the leader of the view is correct and the GST happened before the start of the view).[1] The view change is defined as the extra steps taken in views $v \geq 2$ compared to the protocol for $v = 1$.

3.1.3 Roadmap

In Section 3.2, we describe the formal system model. In Section 3.3, we recall the protocol of [1, 32], which we denote as TH1. Then, in Section 3.4, we explain how to safely augment this protocol with a fast track using the abstraction of a *Proof-of-Exclusivity (PoE)*. In Section 3.5, we describe an efficient protocol for constructing such a proof. In Section 3.6 then Section 3.7, we demonstrate an accountability mechanism that always enables to get this proof "for free", unless the first leader is caught cheating. As a result, plugging PoEs in the upgrade Hotstuff-2 [106] of TH1, ***preserves their responsiveness unless a leader is caught cheating***. In Section 3.8, we overview the related work. We conclude the chapter in Section 3.9 with a discussion on the directions for future work.

3.2 Model and Definitions

In this section, we define the formal system model for our protocols.

Notation. The size of a finite set E is denoted $|E|$. The size of a bitstring b is denoted $|b|$, and the empty string is denoted $\perp$. $\mathbb{N} := \{0, 1, 2, \dots\}$ denotes the set of non-negative integers, of which the positive ones $\mathbb{N}^* := \{1, 2, \dots\}$. Strings of characters are denoted in quotes, as: "string". To *multicast* a message means to send it to every party.

Partially-synchronous network and corruptions. We consider a set $\mathcal{P} = \{P_1, \dots, P_n\}$ of probabilistic polynomial time (PPT) machines connected by pairwise authenticated channels. We denote them as *parties*. We consider a PPT machine, denoted as the *Environment* $\mathcal{E}$, which can read all messages sent and, without further specification, alter, reroute, drop, delay or replay them. $\mathcal{E}$ has full control of up to t parties, where t is a parameter known as *corruption threshold*. We denote them as maliciously *corrupt*, also known as *Byzantine*. For simplicity, we consider the maximal corruption threshold, i.e., we assume that the number of parties is $n = 3t + 1$ and that corruptions happen at the beginning of the execution.[2] The remaining (at least $2t+1$) parties are said to be *honest*. At some point in time denoted as GST [60], $\mathcal{E}$ commits to delivering from now on all messages sent within a fixed delay δ. Both GST and δ are arbitrarily set by $\mathcal{E}$ in every execution, and are not disclosed to the honest parties. However, there is a fixed upper bound $\Delta \geq \delta$ which holds for any execution, and which is publicly known in advance. In some executions, Δ may be much larger than the actual message delay δ. If GST $= 0$, then this means that the δ delay holds since the beginning of the execution.

Views and leaders. For simplicity, we assume a global clock that publicly ticks positive integers in increasing order, starting from 1 at the beginning of the execution. A view v is the timeframe between the ticks of v and of $v + 1$. Along with each new tick v, the clock also designates a party denoted L_v–the *leader of phase* v. For simplicity of the presentation, we assume that exactly one leader L_v is designated per phase v, and that its identity is made public to all parties as soon as the clock ticks v. The simplest implementation is the one of [60], in which parties have weakly synchronized clocks and leaders are predetermined in round robin order (see [103] for a state-of-the-art implementation). Our results carry over to the model with a more general abstraction denoted as "pacemaker", introduced

[1] The motivation for this latency measure is that, in most protocols considered, along with a decision, honest parties output a *decision certificate*, which can be independently verified by a client.

[2] Neither of these assumptions is necessary. We only make them for simplicity of the presentation.

in [142]. See also [30, 31, 114] for more flexible abstractions and efficient implementations of such mechanisms. Importantly, our protocols remain safe even if the delivery of clock ticks to parties are arbitrarily scheduled, and even if parties are notified of different leaders in a view.

Bulletin board PKI setup and threshold signatures. In this chapter, the only cryptographic primitive needed could be formalized in a broad sense as a non-interactive threshold signature scheme [24, 124, 131]. Precisely, we will need the following algorithms.

First, a local algorithm enabling each party to individually generate a pair of keys: a private signature key, and a public verification key. We assume that every party published a public verification key on a public bulletin board, denoted as PKI.

On input its signature key and any message m, party P_i can produce with SIGN what is denoted as a signature $\sigma_i(m)$ on m. There is a public algorithm which checks the validity of σ_i against the verification key of P_i and m. We will dub a valid signature on some message m, with respect to the verification key of a party, as a signature *issued* by this party.

We require the standard existential unforgeability property of digital signature schemes. Then, for any integer k, there is a public algorithm, denoted AGGREGATE_k that, for any given message m, takes as input any set of k valid signatures (σ_i), issued by any k distinct parties, on this same message m, then outputs what is denoted as a *threshold signature* σ on m. There is a public verification algorithm that checks the validity of any threshold signature σ on any m.

The signing, aggregation, and verification algorithms, together, have the property that any polynomial adversary learning up to $k-1$ secret keys, and making oracle queries to obtain partial signatures issued by any of the remaining $n-(k-1)$ keys, cannot possibly forge a valid signature on some message m unless it has already made an oracle query on m. They also have the robustness property that an adversary, even learning all the secret keys, cannot produce k valid signature shares of some m such that the AGGREGATE_k of them would fail to output a valid threshold signature on m.

One possible instantiation for the threshold signature scheme is BLS multi-signatures [25]. In this scheme, the identities of the k signers must be included in the signature σ, which, in our setting, can be represented as n bits. Another possible instantiation is [16, §5], which has the advantage that the identities of the signers are not required for verification, but the signature itself has the size proportional to $\log(n)$. The two schemes have the same, constant verification complexity.

Finally, note that threshold signatures are also implementable under a trusted setup, which we will not assume in this chapter. In this category we have the scheme based on RSA [124] and the one based on BLS [131]. The trusted setup brings the benefits that these signatures have constant bitsize and do not require knowledge of the k signers. These schemes, however, require distinct setups and algorithms for every distinct k. In this chapter, we will consider three values of k: $t+1$, $2t+1$ and $3t+1$.

BFT. Let us consider any public efficiently computable predicate $\mathsf{ExtValid} : \{0,1\}^* \to \{\mathsf{true}, \mathsf{false}\}$. We denote $\mathcal{X} := \{x \in \{0,1\}^* \mid \mathsf{ExtValid}(x) = \mathsf{true}\}$ the set of *externally-valid* values. We assume that each party starts the protocol with an input from $\mathcal{X}$.

Definition 3.1 (BFT)**.** *A* partially synchronous leader-based Byzantine fault-tolerant consensus with external validity*, or* BFT *for short, is a protocol in which each party outputs at most one value (we say that the party* decides *the value) such that the following properties hold:*

Consistency: *no two parties decide different values;*

External Validity: *if a party* decides x*, then* $x \in \mathcal{X}$*;*

Termination: *every honest party* decides *a value in any execution in which there is a sufficiently long normal view* v*.*[3]

Latency and authenticator complexity. Here, we formally define the metrics by which we evaluate the protocols. Neglecting the computation time, the delays are measured in terms of Δ and δ. Following [73, 142], authenticator complexity is defined as the total number of authenticators sent over the

[3]Recall that a view is said to be normal iff L_v is honest and GST happens before the start of the view. The exact definition of "sufficiently long" depends on the protocol. Typically, it is a function of δ and/or Δ.

network, where an authenticator is either a message authentication code (implicitly appended to each message), a signature, or a threshold signature.

The majority of protocols that we consider furthermore enforce that, if a party decides some value x, then it must be the case that it also learned a cryptographic proof, which we designate as a decision certificate, that confirms that x is indeed the only possible output value and that can be checked by any (possibly, external) party. Thus, we will usually measure the delays only up to the point when one honest party decides on a value (as opposed to waiting for all honest parties to decide).

We say that a BFT protocol has a *slow track latency* D_{slow} if, assuming GST $= 0$ and the first leader is honest, L_1 decides within the delay D_{slow} since the start of the protocol. Moreover, we say that a BFT protocol has a *fast track latency* D_{fast} ($D_{\text{fast}} < D_{\text{slow}}$) if, under the additional assumption that all parties are honest, L_1 decides within the delay D_{fast}. Finally, a protocol with a slow track latency D_{slow} has the *view change latency* D_{vc} if it guarantees that, in any normal view $v \geq 2$, the leader of the view L_v decides within the delay $D_{\text{slow}} + D_{\text{vc}}$.

Similarly, we define the *authenticator complexity of a view change* of a BFT protocol as a difference between the authenticator complexity in a view $v \geq 2$ and the authenticator complexity in the first view.

Our protocol stays safe in relaxed models allowing players to be aware of possibly different leaders for the same view, or, not having synchronized view numbers. It still has liveness in this adversarial setting, provided a narrowing of the definition of a normal view.

3.3 Baseline

The protocol of [1], which we dub as TH1, is described in Algorithm 3.2 with the description of the used data structures in Algorithm 3.1.

In Algorithm 3.2, actions are specified with respect to the current view number v. The last action (tagged with $*$) is not conditional to the view number. The actions specific to the first view ($v = 1$) are further highlighted in blue. Parties perform actions as soon as they can, in any order. However, the actions are numbered to reflect their causal dependency in an execution or the protocol. For example, with respect to a given view v, if leader v completes action 3, then it must be the case that at least $t+1$ honest parties completed action 2. Some actions (namely, 1, 3, and 5) are taken only by the leader.

Every view follows the same pattern. The leader selects a value x, in a way that will be detailed later, and sends x with some extra cryptographic metadata to the parties in a prepare message. Upon receiving a prepare message from the leader for the first time, every party verifies the correctness of the metadata and sends back to the leader a signed lock vote message vouching for the value x received in the prepare message. Upon receiving $2t + 1$ lock vote messages vouching for the same value x, the leader aggregates the $2t + 1$ signatures into what we denote as a lock certificate for the value x, which it multicasts to the parties.

Note that the $t < n/3$ assumption implies that no two lock certificates with the same view number for two distinct values $x \neq x'$ can possibly be created. Upon receiving a lock certificate vouching for some value x, a player sends back to the leader a signed vote vouching for this value, in the form of a dec vote message. Upon receiving $2t + 1$ dec vote messages vouching for the same value x, the leader aggregates the $2t + 1$ signatures into what we denote as a decision certificate and multicasts it to the parties.

At this point, we say that we have reached a decision for the value x. Indeed, any honest party which forms or receives a decision certificate for a value x must automatically decide x. Thus, any entity external to the system, upon receiving a decision certificate for a value x, has the guarantee that x has been or will be decided by every party.

Now, since each leader of a view can do these operations, we need to enrich the protocol with an additional safeguard mechanism which guarantees that no two decision certificate can ever be created for two distinct values. Following the terminology of [142], we denote the following mechanism as "view change".

All view change implementations since [60] have in common that every party keeps in memory the lock certificate associated to the highest view number that it ever received or created. We then say that the party is "locked" on the value x vouched by this lock certificate. More specifically, the protocol TH1 follows the rule of [1, 32, 60], which is that a party locked on some value x with view number w, refuses to vote for any other value $x' \neq x$ unless it comes with a lock certificate associated to a higher view number $w' > w$. Upon receiving or forming such a lock certificate associated to a higher view number,

the party releases its lock on the value x and becomes instead locked on the value x'. This mechanism maintains the safety invariant that, if a value is decided in a view, then no lock certificate for any other value can ever be created in any higher view.

To maintain liveness of TH1 with small communication, every view begins with an initial step in which all parties send to the new leader the lock certificate on which they are locked, if any, or their input value otherwise. For consistency of notations, we denote this latter case as a *degenerate* lock certificate $\mathsf{lc}[0, x] = x$. Then, after the maximum network delay Δ has elapsed, the leader multicasts its prepare, to which it appends its highest lock certificate. The liveness is guaranteed by the fact that, after GST, it must be the case that the leader received the highest lock certificate of every honest party. Thus, every party will agree to vote on the value contained in the prepare message.

Theorem 3.2 ([1]). *The protocol TH1 of Algorithm 3.2 is a BFT with normal-case latency 4δ, view change latency Δ, and authenticator complexity $O(n)$.*

- **lock certificate** is the data structure of triples (w, y, σ), where: $w \in \mathbb{N}$ is called *locked view number* and $y \in \mathcal{X}$ is called *locked value*.
  - *If* $w \geq 1$, then σ is a $(2t+1)$-threshold signature on the triple ("lock vote", w, y).
  - *Else if* $w = 0$, then $\sigma = \perp$. In this latter case, we make the identification of the lock certificate with the x itself, and dub such lock certificate as *degenerate*.

  For brevity, we use $\mathsf{lc}[w, y]$ to denote a lock certificate for view w and value y, omitting σ.
- **high lock$_i$** is a local variable maintained by every party P_i, equal to the lock certificate $\mathsf{lc}[w_i, y_i]$ such that w_i is the *highest locked view number* for which P_i ever created or received a lock certificate. Thus, P_i replaces it each time it creates or receives a lock certificate with a strictly higher locked view number.
  - In the beginning of the protocol, every P_i initializes high lock$_i$ with the (degenerate) lock certificate $\mathsf{lc}[0, y_i] = y_i$, where y_i is P_i's input to the BFT protocol. Recall that no threshold signature is required in this case.

  On the other hand, lock certificates for higher view numbers are hard to forge, because of the threshold signature required.
- **report** is a data structure consisting of a pair $(v, \mathsf{lc}[w, y])$, where $v \in \mathbb{N}^*$, such that the condition $w < v$ is satisfied.
- **prepare** is a data structure consisting of a pair $(v, \mathsf{lc}[w, y])$, where $v \in \mathbb{N}^*$, such that the condition $w < v$ is satisfied.
- **lock vote(v, x)** / **dec vote(v, x)** are both data structures consisting of a triple $(v \in \mathbb{N}^*, x \in \mathcal{X}, \sigma_i)$, where σ_i a signature of party P_i on the triple ("lock vote"/"dec vote", v, x). P_i is denoted the *issuer*. We often omit the σ_i for brevity.
- A decision certificate **dec cert(v, x)**, where $v \in \mathbb{N}^*$ and $x \in \mathcal{X}$, is a $(2t+1)$-threshold signature on the triple ("dec vote", v, x).

Algorithm 3.1: Data Structures for TH1

3.4 Fast Track from a Black-Box PoE

3.4.1 Overview

As was first noted in [91] and later exploited in [59, 74, 90, 109], consensus with $n = 3t + 1$ parties can be reached much faster, in just 2δ, in the optimistic case when all parties are honest and the network is synchronous from the beginning of the protocol. The way it is done is pretty simple. The lock votes in the first view now serve a second purpose: if the leader of the first view manages to collect n such votes for the same value x, it creates a *fast decision certificate* for x out of these n votes after just 2 message delays. The leader multicasts it to the parties, any party receiving a *fast decision certificate* for some

The following steps are performed by each party as soon as it is in view v.

0. Report If $v = 1$, skip and go directly to step **1**.
Else if $v \geq 2$*:* Every party P_i, denoting by $\mathsf{lc}[w_i, y_i]$ its $\mathsf{high\,lock}_i$, sends to the leader L_v a message $\mathsf{report}\big(v, \mathsf{lc}[w_i, y_i]\big)$.

1. Prepare *If* $v = 1$, L_1 multicasts $\mathsf{prepare}\big(1, \mathsf{lc}[0, y_{L_1}]\big)$, where $\mathsf{lc}[0, y_{L_1}] = y_{L_1}$ is the input of L_1 to the BFT.
Else if $v \geq 2$*:* Leader L_v receives report messages during the delay Δ since the beginning of the view v. When Δ is elapsed, denoting by $\mathsf{lc}[w_{L_v}, y_{L_v}]$ its $\mathsf{high\,lock}_{L_v}$, L_v multicasts $\mathsf{prepare}\big(v, \mathsf{lc}[w_{L_v}, y_{L_v}]\big)$.

2 Lock Vote Every party P_i, upon receiving a message of the form $\mathsf{prepare}\big(v, \mathsf{lc}[w, y]\big)$ from L_v for the first time: If its highest locked view number w_i is lower than or equal to w, then it replies with a $\mathsf{lock\,vote}\big(v, y\big)$ signed with its signature σ_i.

3. Lock Certificate Upon receiving $\mathsf{lock\,vote}\big(v, x\big)$ from $2t + 1$ distinct issuers, the leader L_v AGGREGATES_{2t+1} their signatures to form a $\mathsf{lc}[v, x]$, which it multicasts to the players.

4. Decision Vote Upon receiving a $\mathsf{lc}[v, x]$ from leader L_v, a party P_i replies with a signed $\mathsf{dec\,vote}(v, x)$.

5. Decision Certificate Leader L_v, upon receiving $2t+1$ $\mathsf{dec\,vote}(v, x)$ from distinct issuers, AGGREGATES_{2t+1} the signatures into a $\mathsf{dec\,cert}(v, x)$, which it multicasts to the players.

$*$ **Decision** Upon forming or receiving a $\mathsf{dec\,cert}(v', x)$ for any $v' \in \mathbb{N}^*$, a party decides x (and continues the protocol).

Algorithm 3.2: TH1: Non-responsive BFT of [1]

x immediately decides x. When there is a fast decision certificate for some value x, we say that x is *committed on the fast track*.

Since an honest party only reacts to the first prepare message it receives from the leader, it will never issue $\mathsf{lock\,vote}(1, x)$ and $\mathsf{lock\,vote}(1, x')$ where $x \neq x'$. Hence, it is easy to see that, due to simple quorum intersection, the leader of the first view will not be able to obtain two conflicting decision certificates (be they fast or not).

However, we also need to preserve the consistency property of BFT in views higher than 1. Namely, we need to also guarantee that, if some value is decided in the first view through the fast track, no other value can be decided in higher views. To this end, we will define a special kind of a cryptographic proof, called *Proof of Exclusivity* (or *PoE* for short). Intuitively, $\mathsf{PoE}(x)$ allows the leader of a higher view L_v, $v \geq 2$, to convince other parties that no value other than x could be committed on the fast track. We provide a formal definition of PoE in Section 3.4.2. To guarantee consistency, in views other than the first one, honest parties only issue decision votes for the values for which a PoE was created, if any.

Prior protocols [2, 5, 74, 90, 94, 109] implicitly implemented a Proof of Exclusivity (without actually defining it) by forwarding $2t + 1$ signed messages reporting values voted in the fast track. Thus, they all have quadratic authenticator complexity in the view change.

3.4.2 Definition of Proof of Exclusivity (PoE)

We now define a type of protocol called *Proof of Exclusivity (PoE) protocol* that can be used to produce a PoE. In a PoE protocol, every honest party P_i has one single input in $\mathcal{X} \cup \{\perp\}$. In addition, one of the players, denoted as L, is publicly designated to play a special role of a *prover*. Apart from its input to the PoE protocol, the prover also has a default valid value $x_{default} \in \mathcal{X}$. The goal of a PoE protocol is for L to obtain a valid value $x \in \mathcal{X}$, satisfying the predicate that no other value x' can possibly be the input of all honest parties, and a publicly verifiable proof that this predicate holds on x.

Definition 3.3. *A value* $x \in \mathcal{X}$ *satisfies the* exclusivity predicate *if there is* no *valid value* $x' \in \mathcal{X}$ *such that* $x' \neq x$ *and all honest parties have input* x'.

Let $x \in \mathcal{X}$*. We say that a bitstring, denoted* $\mathsf{PoE}(x)$*, is a (non-interactive)* **Proof of Exclusivity** *of* x*, if it is such that any verifier, possibly external, can efficiently check on* $\mathsf{PoE}(x)$ *that* x *satisfies*

exclusivity.

We call a value $x \in \mathcal{X}$ *unanimous* iff all honest parties have x as their input to PoE. Hence, the exclusivity predicate can be restated as: "only x can *possibly* be unanimous". Additionally, we use notation $\mathsf{PoE}(\perp)$ to denote a bitstring that can serve as $\mathsf{PoE}(x)$ for any valid value $x \in \mathcal{X}$. For example, it is easy to see that a $(t+1)$-threshold signature on the statement "my PoE input is $\perp$" can serve as a $\mathsf{PoE}(\perp)$.

Definition 3.4. *A protocol between $n = 3t+1$ parties, of which a designated prover L, is denoted as a* PoE protocol *if it guarantees that, if it is started after GST and the prover is honest, then the following happens in finite time:*

- *At some point, L outputs a valid value $x \in \mathcal{X}$. We denote this action as* $\mathsf{PoEAnticipate}(x)$*;*
- *Then, L outputs a* $\mathsf{PoE}(x)$*. We denote this action as* $\mathsf{PoEOutput}(\mathsf{PoE}(x))$*.*

Let us give the intuition of why and how a PoE protocol is used as a subroutine of the BFT with a fast track sketched in section 3.4.1. At the beginning of a new view v of the BFT, parties start an instance of a PoE protocol with the leader of this view $L := L_v$ acting as the prover. Each party P_i sets its input to the PoE protocol x_i equal to the valid value received in the prepare message from the first leader L_1, if any, or to $x_i := \perp$ otherwise. Note that, if the input value of the PoE protocol of an honest party P_i is a valid $x_i \in \mathcal{X}$, then P_i could not have cast a lock vote in the first view for any value other than x_i.

In addition, L sets the default valid value $x_{default} \in \mathcal{X}$ equal to its own valid input of the BFT protocol. The purpose of $x_{default}$ is to be used by the prover as the output value in case when all honest parties (including L) have $\perp$ as their input to the PoE protocol.

At the end of the PoE protocol, L_v outputs some value $x \in \mathcal{X}$ satisfying the exclusivity predicate and, possibly later, a publicly verifiable proof of this predicate, denoted $\mathsf{PoE}(x)$. By definition, $\mathsf{PoE}(x)$ proves that no other value $x' \neq x$ could possibly be such that all honest parties cast a $\mathsf{lock\,vote}(1, x')$. This implies that no fast decision certificate can ever be created for any $x' \neq x$. Thus, $\mathsf{PoE}(x)$ is exactly the proof needed by L_v in our short description of the BFT protocol in section 3.4.1.

The time since the start of the PoE protocol and until the PoEAnticipate event is called *the anticipation latency* of the PoE protocol. This metric affects the latency of the BFT protocol because, as we will see, a leader can propose a value x as soon as it receives the notification $\mathsf{PoEAnticipate}(x)$.

Similarly, the time since the start of the PoE protocol and until the PoEOutput event is called *the output latency* of the PoE protocol. However, since the parties need to know the anticipated value x already at the time when they issue lock vote vouching for this value, but need not verify the $\mathsf{PoE}(x)$ up until the moment when they issue dec vote, as long as output latency is at most anticipation latency plus 2δ, it does not affect the overall latency of the BFT protocol.

3.4.3 Generic BFT with fast track from a black-box PoE protocol

The protocol is described in Algorithm 3.4 and uses the additional or modified data structures as presented in Algorithm 3.3. The notations, in terms of numbered steps, still denote actions which can safely be taken in any order, and, for liveness, are meant to be taken as soon as allowed. We further highlighted with the tag [– fast track] some additional steps which are related to the fast track, although they can be taken in parallel with other steps. Notice that the fast track impacts the slow track, e.g., in steps 0 and 1 for $v \geq 2$, parties must now run in the background a PoE protocol.

Theorem 3.5. *When instantiated with a PoE protocol with anticipation latency Δ and output latency at most $\Delta + 2\delta$, the protocol of Algorithm 3.4 is a BFT with normal-case latency 4δ, fast-track latency 2δ, view change latency Δ, and $O(n)$ authenticator complexity (cf Table 3.1).*

3.4.4 Relationship to prior protocols

In all previous BFT protocols with a fast track, in a view-change, when players received a prepare for some value x not in a (non-degenerate) lock certificate, then they refused to cast a lock vote for x until they received a $\mathsf{PoE}(x)$. But if we apply this rule to our BFT, we will have the following extra latency. Indeed, our PoE protocol in Section 3.5 has worst-case output latency of $\Delta+2\delta$. This is why, in Algorithm

- A **fast dec cert(x)** is a pair $(x, \sigma^{\text{fast}})$ where σ^{fast} is a n-threshold signature on the pair ("lock vote", $1, x$).
- A **lock certificate** $\mathsf{lc}[w, y]$ is as in TH1, but, if $w \geq 1$, it must also contain a PoE on the locked value in order to be considered valid. Formally, for $w \geq 1$, $\mathsf{lc}[w, y]$ is a tuple of the form $\big(w, y, \sigma, \mathsf{PoE}(y)\big)$ where σ is a $(2t+1)$-threshold signature on the triple ("lock vote", w, y). As in TH1, $\mathsf{lc}[0, y] = y$ and does not require either a threshold signature nor a $\mathsf{PoE}(y)$.

Algorithm 3.3: Data Structures of Generic BFT with a Fast Track. Data structures not displayed are the same as in TH1 (Algorithm 3.1)

0. Report If $v = 1$, skip and go directly to step **1**.
Else if $v \geq 2$: Every party P_i, denoting by $\mathsf{lc}[w_i, y_i]$ its high lock$_i$, sends to the leader L_v a message $\mathsf{report}\big(v, \mathsf{lc}[w_i, y_i]\big)$.

- Furthermore, if $w_i = 0$, P_i initiates a PoE protocol with leader $L := L_v$, with input equal to the value $x_i \in \mathcal{X}$ received in the prepare message from the first leader L_1, if any, or $\perp$ otherwise. If P_i is the leader (i.e., $P_i = L_v$), it sets $x_{default}$ in the PoE protocol equal to its input value in the BFT.

1. Prepare *If* $v = 1$: L_1 multicasts $\mathsf{prepare}\big(1, y_{L_1}\big)$, where y_{L_1} is its input to BFT.
Else if $v \geq 2$: Leader L_v receives report messages during delay Δ since the beginning of the view v. When Δ is elapsed, denoting by $\mathsf{lc}[w_{L_v}, y_{L_v}]$ its high lock, leader L_v:

- *If* $w_{L_v} = 0$, L_v waits for $\mathsf{PoEAnticipate}(x)$ and multicasts $\mathsf{prepare}\big(v, \mathsf{lc}[0, x]\big)$.
- *Else if* $w_{L_v} \geq 1$, L_v multicasts $\mathsf{prepare}\big(v, \mathsf{lc}[w_{L_v}, y_{L_v}]\big)$.

2. Lock Vote Every party P_i, upon receiving for the first time a $\mathsf{prepare}\big(v, \mathsf{lc}[w, y]\big)$ from L_v. If its highest locked view number w_i is lower than or equal to w, then it replies with a signed $\mathsf{lock\,vote}\big(v, y\big)$.

[3. – fast track] Fast Decision Certificate If $v = 1$: upon receiving $\mathsf{lock\,vote}\big(1, y\big)$ issued by all $n = 3t+1$ players, L_1 AGGREGATES$_n$ their signatures into a $\mathsf{fast\,dec\,cert}(x)$ and multicasts it to the parties.

3. Lock Certificate Upon receiving $2t+1$ lock votes for some value x, leader L_v AGGREGATES$_{2t+1}$ their signatures. Then, it forms and multicasts $\mathsf{lc}[v, y]$. To this end, in addition to the threshold signature, it will need a $\mathsf{PoE}(x)$. Let $\mathsf{lc}[w, y]$ be the lock certificate L_v multicasted on step 1 (Prepare).

- *If* $w = 0$ *and* $v = 1$, the $(2t+1)$-threshold signature on ("lock vote", $1, y$) serves the purpose $\mathsf{PoE}(y)$;
- *Else if* $w = 0$ *and* $v \geq 2$, L_v waits for the event $\mathsf{PoEOutput}(\mathsf{PoE}(y))$ and takes the PoE from it;
- *Else if* $w \geq 1$, L_v takes $\mathsf{PoE}(y)$ from $\mathsf{lc}[w, y]$.

4. Decision Vote Upon receiving a $\mathsf{lc}[v, x]$ from leader L_v (so, including a $\mathsf{PoE}(x)$ attached), a party P answers by a signed dec vote (v, x).

5. Decision Certificate Leader L_v, upon receiving $2t+1$ dec votes for the same (v, x), AGGREGATES$_{2t+1}$ the signatures into a $\mathsf{dec\,cert}(v, x)$ and multicasts it to the parties.

$*$ **Decision** Upon receiving a $\mathsf{dec\,cert}(v', x)$ for any $v' \in \mathbb{N}^*$, a party decides x (and continues the protocol).

[$*$ – fast track] Fast Decision Upon receiving a $\mathsf{fast\,dec\,cert}(x)$, a party decides x (and continues the protocol).

Algorithm 3.4: Generic BFT with a Fast Track

3.4, we introduced a structural modification of possibly independent interest. Namely, parties now cast a lock vote for a value x, *even if* they did not receive yet a PoE(x). Despite this relaxation, we preserve safety since parties cast a dec vote only if the lock certificate of the current view, which they receive from the leader, is appended with a PoE for the value enclosed. A lock certificate without an appropriate PoE is considered as invalid and completely ignored (it does not unlock *parties*, nor does it change their high lock).

3.5 Big Buckets PoE

In this section, we provide a PoE protocol with linear authenticator complexity, constant size proof, and anticipation latency 1Δ. When we integrate it in the protocol described in Section 3.4.3, we obtain a BFT protocol with the properties summarized in Table 3.1. However, to obtain a protocol that would perform best in practice, the optimizations from Section 3.7 should be applied in order to reduce the hidden constants in the asymptotic complexities.

0. Report Every player P_i sends its input x_i to the prover L in a signed message poe report(x_i).

1. Timely output *or* Anticipate and Request L waits the time interval Δ. When Δ is elapsed, if the prover received at least $2t+1$ poe reports:

- *If* L received at least $t+1$ poe reports for some value $x \in \mathcal{X} \cup \{\perp\}$, then it AGGREGATES$_{t+1}$ the signatures from these reports. If $x \neq \perp$, L triggers PoEAnticipate(x). Otherwise, it triggers PoEAnticipate($x_{default}$). In any case, it also triggers PoEOutput(σ), where σ is the threshold signature and terminates the protocol.
- *Otherwise,* L triggers PoEAnticipate($x_{default}$) and partitions the interval $[0;\infty)$ into k (up to 5) consecutive intervals $B_1, \ldots, B_k$ (called "buckets") as described in the proof of Theorem 3.10 and multicasts message bb-request($b_1, \ldots, b_{k-1}$), where $b_j = \min(B_{j+1})$.

2. Confirm Upon receiving bb-request($b_1, \ldots, b_{k-1}$) from L, each party P_i sends to the prover a poe confirm message with up to k signatures: for each $j \in \{1, \ldots, k\}$ such that $x_i \notin [b_{j-1}; b_j)$ (letting $b_0 = 0$ and $b_k = \infty$), P_i sends to L a signature for the tuple ("poe confirm", $H(b_1, \ldots, b_{k-1})$, "my value is not in bucket j"), where H is a collision-resistant hash function.

3. Output L waits until it receives $t+1$ such signatures for each of the k buckets, AGGREGATES$_{t+1}$ them and triggers PoEOutput($\sigma_1, \ldots, \sigma_k$), where σ_j is the j-th threshold signature.

Algorithm 3.5: Big Buckets PoE

The protocol is summarized in Algorithm 3.5. It starts from each party P_i sending its input value x_i and a signature on the tuple ("poe report", x_i) in a poe report message to the prover (denoted by L). Then L waits for the time period Δ in order to collect poe report messages from all correct processes (assuming GST has happened).

Recall that a value x is called *unanimous* iff it is the PoE input of all honest parties. Let us make a few simple observations.

Lemma 3.6. *If some value x is unanimous, then for any set S of $2t+1$ poe reports issued by different parties, at least $t+1$ reports are for value x.*

Proof. In a set of $2t+1$ parties there must be at least $t+1$ honest parties, which will only report x. □

Lemma 3.7. *If $t+1$ parties report that their PoE input is not x, then x is not unanimous.*

Proof. In any set of $t+1$ parties, there must be at least 1 honest party. If x was unanimous, this player would report that its value is x. □

Corollary 3.8. *If $t+1$ parties report that their PoE input is $x \in \mathcal{X} \cup \{\perp\}$, then no value $x' \in \mathcal{X}$, $x' \neq x$, can be unanimous.*

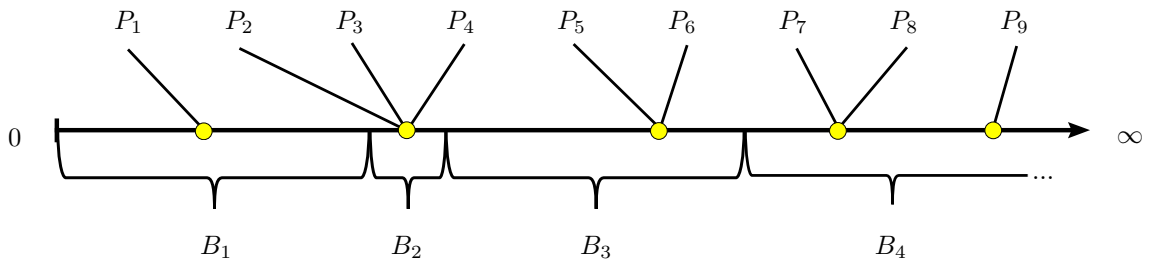

Figure 1: Bucket allocation example with 9 reports and $t = 3$

We now detail the rest of the protocol after Δ. If, by the end of the Δ waiting period, L does not collect at least $n-t = 2t+1$ poe reports, then it can conclude that GST has not yet happened and simply terminate the protocol prematurely (the protocol does not guarantee liveness in this case). Otherwise, it can check if it has collected at least $t+1$ poe reports for some value x. If there is such a value, then, by Theorem 3.8, L can simply AGGREGATE_{t+1} the signatures on ("poe report", x) to obtain $\mathsf{PoE}(x)$ and trigger both $\mathsf{PoEAnticipate}(x)$ and $\mathsf{PoEOutput}(\sigma)$, where σ is the aggregated threshold signature.

Otherwise, by Theorem 3.6, L can conclude that there is no unanimous value and any value can be safely anticipated. In this case, L will trigger $\mathsf{PoEAnticipate}(x_{default})$ and, in the rest of the protocol, will try to produce $\mathsf{PoE}(\perp)$.

The prover will construct $\mathsf{PoE}(\perp)$ by relying on Theorem 3.7: it will create a short proof that, for each value $x \in \mathcal{X}$, there are at least $t+1$ parties voting against x. To this end, L will partition the range of values $\mathcal{X}$ (more precisely, the range of binary representations of values) into a constant number of intervals $B_1, \ldots, B_k$, called *buckets*, such that each bucket will contain values from at most t of the received poe reports. An example of such an allocation can be seen in Figure 1.

As we show in Theorem 3.10, for $n = 3t+1$, just 5 buckets are always sufficient. Hence, a bucket allocation can be encoded with at most 4 binary strings of the same size as the input values: $b_1 = \min(B_2)$, $\ldots$, $b_{k-1} = \min(B_k)$. The prover sends these numbers to the parties and asks them to report to which buckets their input values do *not* belong (by convention, $\perp$ does not belong to any bucket). More precisely, upon receiving from the prover the $k-1$ numbers encoding the buckets, honest party P_i with input value x_i responds with up to k signatures: for each $j \in \{1, \ldots, k\}$ such that $x_i \notin [b_{j-1}+1; b_j)$ (letting $b_0 = 0$ and $b_k = \infty$), P_i sends to L a signature for the tuple ("poe confirm", $H(b_1, \ldots, b_{k-1})$, "my value is not in bucket j"), where H is a collision-resistant hash function.

Assuming that GST happened before the start of the PoE protocol, L must have initially received poe reports from all honest parties and it knows that no bucket contains more than t of the reports. Since each honest party will reply to L, for each $j \in \{1, \ldots, k\}$, L will be able to AGGREGATE_{t+1} a signature for the tuple ("poe confirm", $H(b_1, \ldots, b_{k-1})$, "my value is not in bucket j"). By Theorem 3.7, these k threshold signatures constitute $\mathsf{PoE}(\perp)$ and, hence, $\mathsf{PoE}(x_{default})$ as well.

Theorem 3.9. *The protocol in Algorithm 3.5 is a partially-synchronous non-responsive PoE protocol with $O(n)$ authenticator complexity, proof size $O(1)$, anticipation latency 1Δ, and output latency $1\Delta+2\delta$.*

Proof. If GST happened before the start of the PoE protocol, all honest parties will send poe reports to the prover and, after waiting for the time interval Δ, the prover will trigger PoEAnticipate with some value (either a value contained in $t+1$ poe reports or $x_{default}$). Now, either the prover will output a PoE immediately (by aggregating $t+1$ signatures for the same value) or it will send a bb-request message.

In the latter case, since the prover had taken into account poe reports of all honest parties when it was constructing the buckets, for each bucket B_j, at least $n-2t = t+1$ honest players will send a poe confirm message with signatures that their values are not in B_j and the prover will trigger PoEOutput.

The validity of the output proofs follows directly from Theorems 3.7 and 3.8. □

Finally, let us prove that 5 buckets are always sufficient.

Lemma 3.10. *Given a set S of* poe report *messages from up to $n = 3t+1$ distinct parties, provided that there is no value that received more than t reports, it is always possible to partition the set of integer numbers $[0; \infty)$ into at most 5 consecutive intervals $B_1, \ldots, B_5$ such that, for each j, values from at most t* poe reports *fall into the interval B_j.*

Proof. The following explicit construction ensures our claim. Let $b_0 = 0$, and b_1 be the largest integer such that the number of poe reports for the values in $[b_0; b_1)$ is at most t (or ∞ if this holds for any integer b_1). If $b_1 < \infty$, repeat the procedure for b_2 and the interval $[b_1; b_2)$. Continue until, for some k, $b_k = \infty$. For all $j \leq k$, let B_j be $[b_{j-1}; b_j)$. Note that, since there is no value contained in more than t reports, for all $j \leq k$, the interval $[b_{j-1}; b_j)$ contains a value from at least one poe report. This ensures that the process eventually terminates and $k < \infty$.

Finally, let us prove that $k \leq 5$. Suppose, for contradiction, that $k > 5$. Consider buckets B_1 and B_2. Together, they must cover values from at least $t+1$ poe reports because, otherwise, by construction, all the reports would be covered by B_1. The same reasoning applies to B_3 and B_4. Hence, there are at most $3t+1-2\cdot(t+1) < t$ poe reports remaining for B_5. Therefore, $b_5 = \infty$ and $k = 5$. A contradiction. □

3.6 Achieving Accountably Zero-overhead and Responsiveness

In Section 3.7, we further optimize the fast BFT protocol so that executions fall in either of the two following categories:

1. Either the execution has practically no complexity overhead compared to TH1, meaning no extra generations, retransmissions, or verifications of threshold signatures; Moreoever, the generation of the PoE is then non-interactive and responsive: the leader generates it upon receiving any $2t+1$ poe reports; Hence, it can be plugged in Hotstuff-2 [106] with, accountably, zero latency or complexity overhead.

2. Or an externally verifiable proof that the leader of the first view, L_1, violated the protocol is produced along with a PoE. Such a guarantee to catch, under some conditions, a misbehaving party, is commonly referred to as *accountability*.

Let us convey the idea on a non-optimized version: the first leader signs the proposed value, then players poe report() to next leaders: either their received signed proposal, or $\perp$. As a result, unless a leader is reported equivocating signed proposed values from the first leader, it is guaranteed to receive either $t+1$ reports for the same value x, which then constitutes of PoE(x), or for $\perp$, which constitutes a PoE(any).

3.7 Optimistic zero-overhead PoE and accountability

Our starting observation is that our PoE protocol of Section 3.5 suffers from costly steps in which parties send up to 5 signatures to the leader, which aggregates them into a PoE. Then it sends it back to all parties, which have to verify the 5 threshold signatures. Although these extra steps do not impact the asymptotic complexities nor the latency of BFT, it is still a significant overhead. Thus, we would like to avoid this costly steps whenever possible.

To this end, we start by observing that these costly steps need not be taken if there is at most one valid value $x \in \mathcal{X}$ such that all players report to have either x or $\perp$ as their input to the PoE protocol. We can easily enforce this favorable configuration of inputs to happen in every BFT execution in which the leader of the first view, L_1, did not send equivocating prepare messages (even if GST happened later). To this end, we require that inputs of the PoE protocol are either $\perp$ or carry a signature of L_1 in order to be considered valid by the prover.

Our roadmap now consists of optimizing for this favorable case and forcing it to happen as much as possible.

First, in Section 3.7.1, we provide a further optimization of the PoE protocol of Section 3.5 in the executions where the first leader does not equivocate. Namely, we remove the need for the prover to verify and aggregate partial signatures on the poe report messages by reusing the signatures produced as part of the baseline protocol TH1. We thus denote this case as a the *optimistic zero-overhead track* of the (improved) PoE protocol. If, however, the prover detects an equivocation of the first leader L_1, it executes Big Buckets PoE as described in Section 3.5: we denote this as the *fallback track*.[4]

Then, in Section 3.7.2, we will discuss how to use accountability mechanisms to make sure that the optimistic zero-overhead track is used as much as possible.

[4]In principle, any other PoE construction can be used as a fallback, potentially with some minor adaptations.

3.7.1 Optimistic PoE protocol with zero–overhead when integrated into BFT

The optimized PoE protocol is described in Algorithm 3.6. The model of this new optimized PoE protocol is slightly enriched, in that the inputs of honest parties which are not $\perp$, are now assumed to carry a signature of L_1. We leave implicit in the PoE protocol that the prover L ignores all reported inputs which do not come appended with a signature of L_1.

Intuitively, the *zero–overhead* track is triggered by the prover if the reported inputs are in the favorable configuration: they are all either $\perp$ or equal to one unique valid value $x \in \mathcal{X}$. Upon observing such a configuration after waiting Δ, the prover has the certainty that, if GST happened before the start of the view, since the reported inputs contain in particular all those of honest parties, it will be able to successfully obtain at least $2t + 1$ signed declarations "my input is either x or $\perp$" if it requests it to parties.

However, in order to satisfy the zero-overhead claim, these declarations are signed with the same signature as the lock votes. More specifically, on step [2. Lock Vote] of Algorithm 3.4, instead of simply signing the tuple ("lock vote", v, x), the parties add statement "my PoE input is either x or $\perp$" to the tuple. Then a single threshold signature is sufficient to serve both purposes.

0. Report Every player P_i sends its input x_i to the prover L in a signed message poe report(x_i).

1. Anticipate and Request L waits until the time interval Δ is elapsed. Then, if L received at least $2t + 1$ poe reports, it continues depending on those two alternatives:

- [zero-overhead] *If* there exists a value $x \in \mathcal{X}$ such that all $2t+1$ reported values are in $\{x, \perp\}$: L triggers PoEAnticipate(x) and multicasts zo-request(x). Note that, if all poe reports are for $\perp$, $x_{default}$ can be used as x.
- [fallback] *Otherwise,* execute Big Buckets PoE as specified in Algorithm 3.5 starting from step 1.

2. Zero-Overhead Confirm An honest party P_i, when executing step [2. Lock Vote] of Algorithm 3.4, waits for either zo-request(x) or bb-request($b_1, \ldots, b_{k-1}$) from the leader.

- [zero-overhead] If zo-request(x) is received and the PoE input of P_i is either x or $\perp$, sign the tuple ("lock vote", v, x, "my PoE input is either x or $\perp$").
- [fallback] If a bb-request message is received, continue executing the Big Buckets PoE (Algorithm 3.5) and the BFT protocol (Algorithm 3.4) unchanged.

3. Zero-Overhead Output Leader L_v, upon receiving $2t+1$ identical signatures on the tuple ("lock vote", v, x, "my PoE input is either x or $\perp$") and producing the threshold signature σ in step [3. Lock Certificate] of Algorithm 3.4, trigger PoEOutput((σ, v, x)). Note that v and x are included in the PoE output in order for it to be externally verifiable and satisfy the definition of Section 3.4.2.

Algorithm 3.6: Zero-overhead PoE protocol

3.7.2 Accountability in case of fallback in the PoE

In this section, we will address two issues that may prevent zero-overhead track from being executed:

1. So far, nothing prevents a malicious first leader from equivocating, and thus preventing the execution of the PoE protocol from benefiting from the zero-overhead optimization; and
2. Even if the first leader behaved correctly, nothing prevents a malicious current leader L_v from not taking the optimistic zero-overhead track, and instead triggering parties to take the fallback track of the PoE protocol of Section 3.5, thus making parties spend unnecessary resources.

The first ingredient to address the first issue (1), which we already required in the optimized PoE (Algorithm 3.6), is that the prepare messages of the first leader are now required to be signed in order to be considered as valid (otherwise they are ignored by honest parties). Thus, if an honest current leader

must follow the fallback in the PoE protocol, it must be the case that it was reported conflicting prepare messages signed by the first leader. These conflicting signed messages constitute a *proof of misbehaviour* of the first leader L_1. Thus, an equivocating first leader L_1 takes the risk that an honest leader will later publicly exhibit a proof of misbehaviour, which may trigger some form of punishment (e.g., via reconfiguration[100] or slashing [33]) for L_1. Thanks to this deterrence, it is possible to make sure that the fallback has to be used by honest leaders in very few occasions, which solves (1).

The additional mechanism to address issue (2) is now very simple: upon receiving a request message from the current leader L_v relative to step [2. Confirm] of the fallback track of the PoE, a party checks if this message comes appended with a proof of misbehaviour for the first leader L_1. If not, then the party ignores subsequent messages from L_v. This mechanism does not harm liveness, since, by construction, if an honest leader L_v triggers the fallback then it must be the case that it *has* a proof of misbehaviour.

In conclusion, we achieve the following alternative in *every view* v, *even if* the current leader $L_{v\geq 2}$ is corrupt:

zero–overhead Either the authenticator complexity of the view is equal to the one of TH1 (plus one plain signature appended by L_1 on its prepare messages and n unverified partial signatures on poe report messages);

fallback Or, there exists at least one honest party which receives from the leader L_v a proof of misbehaviour of L_1.

3.8 Related Work

PBFT [40] is considered to be the first practical algorithm solving Byzantine fault-tolerant consensus in partial synchrony. The protocol ensures that a decision is reached within 3 message delays in a normal view with quadratic authenticator complexity.

BFT consensus with linear authenticator complexity. To the best of our knowledge, Hot-Stuff v1 [1], based on [32], is the first partially-synchronous leader-based deterministic Byzantine fault-tolerant consensus protocol with linear authenticator complexity in every view. We denote this protocol as TH1 and use it as the baseline for our protocols. It instructs each new leader to wait for the eventual maximum network delay Δ before taking actions.

In [142], Yin et al. lift this requirement at the expense of an extra round-trip of communication (the proposed protocol needs 3 round-trips in a view before a decision can reached, compared to 2 in TH1).

PnS [8] achieves responsive BFT in 2 round trips, with a PBFT-like view change that requires the parties to send at most ϕ signatures to the leader, where ϕ is the number of views since the beginning of the instance until when the network stabilizes and an honest leader is elected, which is expected to be small in most cases. The computational cost is that of checking one aggregate signature.

Wendy [73] is a BFT protocol which has 2 round-trips in every view in which no party has a higher lock than the leader. When this happens, [73] has 3 round-trips. It has a one-time cost of $2\log(v_{upper})$ authenticators published in the PKI per player and a one-time computation cost per player equivalent to verifying $2n\log(v_{upper})$ aggregate signatures, where v_{upper} is a conservative upper bound on the maximum possible number of views in an execution. In each view, the leader forwards to the parties $O(n)$ bits and one authenticator. The computation complexity in each view is the same as verifying an aggregate signature, plus a multiplicative overhead of $\log(\phi)$ for the number of linear operations in the group.

The protocol of [130] has linear authenticator complexity, takes two round-trips when all parties are locked on the previous view number (denoted as a "happy path"), and 3 round-trips otherwise. Both the protocols Jolteon [71] (implemented in Diem) and [86] have the opposite trade-off: always 2 round-trips, but the authenticator complexity is linear only if all parties are locked on the previous view number, otherwise it is quadratic. Notice that in [86], this quadratic authenticator complexity is because players have to check an aggregate signature of [28], which requires the computation of $2t + 2 = O(n)$ pairings (instead of two, for a multi-signature or a BLS threshold signature).

Fast track in BFT consensus. Kursawe [91] was the first to implement a Byzantine fault-tolerant consensus protocol with an optimistic fast track. The protocol suggested in [91] tolerates up to one-third of Byzantine failures and is able to reach a decision in two message delays when all parties are honest

and the network is synchronous. Otherwise, the protocol falls back to a much less efficient randomized asynchronous consensus protocol.

Later, Martin and Alvisi [109] integrated a fast track into PBFT [40] in a protocol called FaB Paxos. Moreover, they generalized the idea by providing an algorithm that requires $n \geq 3t + 2f + 1$ parties, can tolerate t Byzantine parties, and is able to reach a decision after two message delays when the network is synchronous and at most f parties are Byzantine, thus exposing a fine-grained trade-off between the resilience and the level of "optimism" required for the fast track. The protocol proposed in this chapter can also be easily generalized in a similar way.

As was first noted in [59] and later rediscovered in [5, 94], the resilience of the generalized FaB Paxos can be improved from $3t + 2f + 1$ to $3t + 2f - 1$. These results are mostly relevant for small-scale systems and, thus, target different applications than the ones considered in this chapter.

Zyzzyva [90], UpRight [47], and SBFT [74] are practical systems that build upon the ideas from FaB Paxos [109]. The evaluations in these papers demonstrate that Byzantine consensus protocols with fast track can achieve performance comparable with crash fault-tolerant solutions in the most common case when there are no (or very few) corrupted parties.

Bosco [127] provides an alternative take on the idea of a fast track. The protocol is able to reach a decision after just one message delay in an optimistic case when all honest parties propose the same value. It requires $n \geq 5t + 1$ or $n \geq 7t + 1$, depending on the desired validity property.

To the best of our knowledge, prior to this work, there were no BFT protocols with fast track and linear per-view authenticator complexity.

An efficient randomized round-synchronization protocol [114] may allow to translate a leader-based consensus protocol with linear per-view authenticator complexity to a protocol with expected linear authenticator complexity in the worst case.

Another application of Big Buckets PoE. Since the publication of the first version of this work, the Big Buckets PoE construction has found an independent application in improving the adaptive complexity of synchronous consensus [46].

3.9 Concluding Remarks

While, in terms of practical efficiency, Big Buckets PoE is likely to be optimal, thanks to the optimistic zero-overhead responsive path, it has the added complexity of having the interactive fallback. Hence, non-interactive PoE constructions would be of interest and could be preferable for some practical implementations.

More broadly, we believe that Byzantine fault-tolerant distributed protocols in general and consensus protocols in particular could benefit from more creative applications of cryptography and from expanding the cryptographic tool set. In this chapter, we focused specifically on combining fast track with linear complexity. However, there are likely many other ways to improve protocols by integrating different kinds of ad-hoc cryptographic proofs.

Another likely underutilized tool for distributed protocols is game theory. In this chapter, we rely on it in its simplest form—accountability. We optimize the protocol for the good case of a non-equivocating leader and then punish the leader otherwise. Another common application is to reward parties for following the protocol. A more rogue approach could be trying to harvest the power of free markets by intentionally underspecifying certain parts of the protocols and focusing on aligning the incentives of the participants with the overall performance. If incentives are big enough, this will encourage people to find creative ways to optimize the system.

Chapter 4

Weight Reduction Problems and Their Applications

4.1 Introduction

4.1.1 Weighted distributed problems

Traditionally, distributed problems are studied in the egalitarian setting where n parties communicate over a network and any t of them can be faulty or corrupted by a malicious adversary. Different combinations of n and t are possible depending on the problem at hand, the types of failures (crash, omission, semi-honest, or malicious, also known as Byzantine), and the network model (typically, asynchronous, semi-synchronous, or synchronous). However, for most distributed protocols, t has to be smaller than a certain fraction of n. For example, most practical Byzantine fault-tolerant consensus protocols [39, 40] can operate for any $t < \frac{n}{3}$. We call such models *nominal* and use f_n to denote their *resilience*, i.e., a nominal protocol with resilience f_n operates correctly as long as less than $f_n n$ parties are corrupt, where n is the total number of participants.

However, this simple corruption model is not always sufficient to express the actual fault structure or trust assumptions of real systems. As a result, we see many practical blockchain protocols adopt a more general, *weighted* model, where each party is associated with a real *weight* that, intuitively, represents the number of "votes" this party has in the system. The assumption on the *number* of corrupt parties in this setting is replaced by the assumption that the *total weight* of the corrupt parties is smaller than a fraction f_w of the total weight of all participants. For example, in permissionless systems, the weight can correspond to the amount of "stake" or computational resources a participant has invested in the system and, in the context of managed systems, to a function of the estimated failure probability.

There are two main reasons for adopting the weighted model in the context of blockchain systems. First and foremost, it protects the system from the infamous *Sybil attacks*, i.e., malicious users registering themselves multiple times in order to obtain multiple identities, thereby surpassing the resilience threshold f_n. Second, it is speculated that users with a greater amount of resources (monetary, computational, or otherwise) invested in the system, and consequently a higher weight, will be more committed to the system's stability and less likely to engage in malicious behavior.

4.1.2 Weighted voting and where it needs help

Perhaps, the most prevalent tool used for the design of distributed protocols is *quorum systems* [72, 107, 113]. Intuitively, to achieve fault tolerance, each "action" is confirmed by a sufficiently large set of participants (called a *quorum*). Then, if two actions are conflicting or somehow interdependent (e.g., writing and reading a file in a distributed storage system), then the parties in the intersection of the quorums are supposed to ensure consistency. Thus, many distributed protocols can be converted from the nominal to the weighted setting simply by changing the quorum system, i.e., instead of waiting for confirmations from a certain number of parties, waiting for a set of parties with the corresponding fraction of the total weight. We call this strategy *weighted voting* and it often allows translating protocols

from the nominal to the weighted model while maintaining the same resilience (i.e., $f_w = f_n$) and, in some cases, with virtually no overhead.

However, weighted voting has two major downsides. First and foremost, many protocols rely on primitives beyond simple quorum systems, and weighted voting is often insufficient to translate these protocols to the weighted model. Notable examples include threshold cryptography [24, 54], secret sharing [35, 123], erasure and error-correcting codes [105], and numerous protocols that rely on these primitives.

Another example, relevant to blockchain systems, where weighted voting is typically not sufficient is in Single Secret Leader Election protocols [26, 42, 43, 67]. It illustrates that not all protocols that cannot be converted to the weighted model simply3 by applying weighted voting belong to the categories above and motivates the general approach taken in this chapter.

The second drawback of weighted voting is that it requires a careful examination of the protocol in order to determine whether weighted voting is sufficient to convert it to the weighted model, as well as non-trivial modifications to the protocol implementation. It would be much nicer to have a "black-box" transformation that would take a protocol designed and implemented for the nominal model and output a protocol for the weighted model. In this chapter, we offer both a "black-box" transformation and a set of more efficient "open-box" transformations for a wide range of problems.

4.1.3 Our contribution

Our contribution to the fields of distributed computing and applied cryptography is twofold:

1. We present a simple and efficient black-box transformation that can be applied to convert a wide range of protocols designed for the nominal model into the weighted model. Crucially, one can determine the applicability of the transformation simply by examining the *problem* in question (e.g., Byzantine consensus), instead of the *protocol* itself (e.g., PBFT [40]) and it does not require modifications to the source code, only a slim wrapper around it. The price for this transformation is an arbitrarily small decrease in resilience ($f_w = f_n - \epsilon$, where $\epsilon > 0$) and an increase in the communication and computation complexities proportional to $\frac{f_w}{\epsilon}$.

2. Furthermore, by opening the black box and examining the internal structure of distributed protocols, we discover that by combining our transformation with weighted voting, in many cases, we can obtain weighted algorithms *without* the reduction in resilience ($f_w = f_n$) and with a minor or non-existent performance penalty.

We summarize some examples of our techniques applied to a range of different protocols in Table 4.1. The last two columns of the table give the upper bound on the overhead of the obtained weighted protocols compared to their nominal counterparts executed with the same number of parties. Note, however, that, in many cases, the overhead applies only to specific parts of the protocol, which may not be the bottlenecks. Thus, further experimental studies may reveal that the real overhead is even lower or nonexistent, even with the worst-case weight distribution. Columns "f_w" and "f_n" specify the resilience of the weighted protocols obtained and the original nominal protocols, respectively. As discussed above, in most cases, we manage to avoid sacrificing resilience (i.e., $f_w = f_n$).

Furthermore, the main building block of our constructions, the *weight reduction problems*, may be of separate interest and may have important applications beyond distributed protocols. It is indeed an interesting and somewhat counterintuitive observation that large real weights can be efficiently reduced to small integer weights while preserving the key structural properties. We formally define the three weight reduction problems considered in this chapter in Section 4.2 and present a practical solver called Swiper in Section 4.3.

4.1.4 Empirical study

The performance of the weighted protocols constructed as suggested in this chapter is sensitive to the distribution of the participants' weights. While we provide upper bounds and thus analyze our protocols for the worst weight distributions possible, it is interesting whether such weight distributions emerge in practice.

distributed problem	nominal solutions	weight reduction problem	f_w	f_n	worst-case average comm. overhead	worst-case average comp. overhead
Derived Protocols						
Efficient Asynchronous State-Machine Replication	[50, 57, 88, 111, 128]	WR for RNG WQ for Broadcast	1/3	1/3	$\times 1.33$ for Broadcast $\times 1.33$ for RNG	$\times 3.56$ for Broadcast $\times 1.33$ for RNG
Structured Mempool	[50]	WQ for Broadcast	1/3	1/3	$\times 1.33$ for Broadcast	$\times 3.56$ for Broadcast
Validated Asynchronous Byzantine Agreement	[4, 36]	WR for RNG	1/3	1/3	$\times 1.33$ for RNG	$\times 1.33$ for RNG
Consensus with Checkpoints	[18]	WR for signing	1/3	1/3	$\times 1.33$ for signing	$\times 1.33$ for signing
Linear BFT Consensus Chain-Quality SSLE	[142] [26]	WR (BB)	1/4	1/3	$\times 2.67$	$\times 2.67$
Useful Building Blocks						
Erasure-Coded Storage and Broadcast	[38, 80, 115, 116, 120, 141]	WQ	1/3	1/3	$\times 1.33$	$\times 3.56$
		WR (BB)	1/4	1/3	–	$\times 3$
Error-Corrected Broadcast	[52]	WQ	1/3	1/3	$\times 1.33$	$\times 7.11$
		WR (BB)	1/4	1/3	–	$\times 3$
Verifiable Secret Sharing	[123]	WR	1/3	1/3	$\times 1.33$	$\times 1.33$
Common Coin Blunt Threshold Signatures Blunt Threshold Encryption Blunt Threshold FHE	[37, 119] [24, 124, 129] [54] [27, 85]	WR	1/3	1/2	$\times 1.33$	$\times 1.33$
Tight Secret Sharing Tight Threshold Signatures Tight Threshold Encryption Tight Threshold FHE	See sec. 4.4.3 (this chapter)	WR	1/2	1/2	$\times 1.33$ ($+O(n^2)$ small messages)	$\times 1.33$

Table 4.1: Examples of suggested weighted distributed protocols with the upper bounds on communication and computation overhead compared to the nominal solutions with the same number of participants. See Sections 4.4 to 4.6 for details on how these numbers were obtained. In Section 4.7, we study real-world weight distributions and conclude that, in practice, the overhead should be much smaller. "WR" and "WQ" refer to the weight reduction problems defined in Section 2.5. "WR (BB)" refers to the black-box transformation described in Section 4.4.4.

To study real-world weight distributions, we tested our weight reduction algorithms on the distribution of funds from multiple existing blockchain systems [14, 75, 95, 110] ranging in size from a hundred parties [14, 15] up to multiple tens of thousands [11, 110]. We perform an in-depth analysis in Section 4.7.

Roadmap

The rest of the chapter is organized as follows: we formally define weight reduction problems and state the upper bounds in Section 4.2. We present Swiper in Section 4.3. The proof that it satisfies the stated bounds is delegated to Section 4.8. Sections 4.4 to 4.6 discuss in detail the applications of the weight reduction problems in distributed computing and cryptography. In Section 4.7, we discuss the results of the empirical study performed on real-world weight distributions. We discuss related work in Section 4.11. Section 4.8 presents the formal proofs of the upper bounds. Section 4.9 contains the mixed integer programming formulation of the Weight Restriction problem. We conclude the chapter with the discussion of directions for future work in Section 4.12.

4.2 Weight reduction problems

Let us define the key building block to our construction, the *weight reduction problems*: a class of optimization problems that map (potentially large) real weights $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$ to (ideally small) integer weights $t_1, \ldots, t_n \in \mathbb{Z}_{\geq 0}$ while preserving certain key properties. For convenience, we use the word *"tickets"* to denote the units of the assigned integer weights, i.e., if $t_1, \ldots, t_n$ is the output of a weight reduction problem, we say that party i is given t_i *tickets*.

NOTATION

To avoid repetition, throughout the rest of the chapter, we use the following notation:

1. $[n] := \{1, 2, \ldots, n\}$
2. for any $S \subseteq [n]$: $w(S) := \sum_{i \in S} w_i$
3. for any $S \subseteq [n]$: $t(S) := \sum_{i \in S} t_i$
4. $W := w([n]) = \sum_{i=1}^{n} w_i$
5. $T := t([n]) = \sum_{i=1}^{n} t_i$

4.2.1 Weight Restriction

The first weight reduction problem is *Weight Restriction* (or simply WR). It is parameterized by two numbers $\alpha_w, \alpha_n \in (0, 1)$ and requires the mapping to preserve the property that any subset of parties of weight less than α_w obtains less than α_n tickets. More formally:

PROBLEM 1 (WEIGHT RESTRICTION)

Given $\alpha_w, \alpha_n \in (0, 1)$ and $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$ such that $W \neq 0$ as input, find $t_1, \ldots, t_n \in \mathbb{Z}_{\geq 0}$ such that T is minimized, subject to the following restriction:

$$\forall S \subseteq [n] \text{ s.t. } w(S) < \alpha_w W : t(S) < \alpha_n T$$

In Section 4.4, we apply Weight Restriction to implement the black-box transformation announced in Section 4.1.3 as well as weighted versions of secret sharing and threshold cryptography with different access structures. In Section 4.8, we prove the following theorem:

Theorem 4.1 (WR upper bound)**.** *For any $\alpha_w, \alpha_n \in (0, 1)$ such that $\alpha_w < \alpha_n$ and any $w_1, \ldots, w_n$: there exists a solution to the Weight Restriction problem with $T \leq \left\lceil \frac{\alpha_w(1-\alpha_w)}{\alpha_n - \alpha_w} n \right\rceil$*

To make sense of this expression, note that: (1) it is proportional to n; (2) it is inversely proportional to the "gap" between α_w and α_n; (3) the numerator $\alpha_w(1 - \alpha_w)$ is smaller than 1 and, in fact, never exceeds $1/4$. For a fixed α_w, one can see $\alpha_w(1-\alpha_w)$ as the "constant" and $O\left(\frac{n}{\alpha_n - \alpha_w}\right)$ as the "complexity".

4.2.2 Weight Qualification

The next weight reduction problem we study is *Weight Qualification* (or simply WQ). It requires the mapping to preserve the property that any subset of parties of weight greater than β_w obtains more than β_n tickets. In some sense, WQ is the opposite of the Weight Restriction problem discussed above. More formally:

PROBLEM 2 (WEIGHT QUALIFICATION)

Given $\beta_w, \beta_n \in (0, 1)$ and $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$ such that $W \neq 0$ as input, find $t_1, \ldots, t_n \in \mathbb{Z}_{\geq 0}$ such that T is minimized, subject to the following restriction:

$$\forall S \subseteq [n] \text{ s.t. } w(S) > \beta_w W : t(S) > \beta_n T$$

In Section 4.5, we show how to apply Weight Qualification to implement weighted versions of storage and broadcast protocols that rely on erasure and error-correcting codes for minimizing communication and storage complexity.

There exists a simple reduction between WR and WQ:

Theorem 4.2. *For any $\beta_w, \beta_n \in (0, 1)$ and $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$, the following problems are identical:*

1. $\text{WQ}(\beta_w, \beta_n, w_1, \ldots, w_n)$
2. $\text{WR}(1 - \beta_w, 1 - \beta_n, w_1, \ldots, w_n)$

Proof. Let us prove that any valid solution to $\mathrm{WR}(1-\beta_w, 1-\beta_n, w_1, \ldots, w_n)$ is a valid solution to $\mathrm{WQ}(\beta_w, \beta_n, w_1, \ldots, w_n)$. The inverse can be proven analogously. Indeed, if $t_1, \ldots, t_n$ is a valid solution for $\mathrm{WR}(1-\beta_w, 1-\beta_n, w_1, \ldots, w_n)$, then $\forall S \subseteq [n]$ such that $w(S) > \beta_w W$ it holds that $w([n] \setminus S) = W - w(S) < (1-\beta_w)W$. Hence, $t([n] \setminus S) < (1-\beta_n)T$ and $t(S) = T - t([n] \setminus S) > \beta_n T$. □

From Theorems 4.1 and 4.2, we obtain the following:

Corollary 4.3 (WQ upper bound)**.** *For any $\beta_w, \beta_n \in (0,1)$ such that $\beta_n < \beta_w$: there exists a solution to the Weight Qualification problem with $T \leq \left\lceil \frac{\beta_w(1-\beta_w)}{\beta_w - \beta_n} n \right\rceil$*

4.2.3 Weight Separation

Finally, Weight Separation, in a sense, combines WR and WQ: it has two parameters, α and β, and guarantees that any set of weight β receives more tickets than any set of weight α. Intuitively, it is similar to solving $\mathrm{WR}(\alpha, \gamma)$ and $\mathrm{WQ}(\beta, \gamma)$ for some unknown $\gamma \in (0,1)$ *at the same time*, i.e., with just a single ticket assignment.

Problem 3 (Weight Separation)

Given $\alpha, \beta \in (0,1)$ and $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$ such that $W \neq 0$ as input, find $t_1, \ldots, t_n \in \mathbb{Z}_{\geq 0}$ such that T is minimized, subject to the following restriction:

$$\forall S_1, S_2 \subseteq [n] \text{ s.t. } w(S_1) < \alpha W \text{ and } w(S_2) > \beta W\text{: } t(S_1) < t(S_2)$$

In this chapter, we focus primarily on Weight Restriction and Weight Qualification because they are sufficient for most applications and, being less restrictive on the ticket assignment, permit more efficient solutions. However, for completeness, we also provide an upper bound on Weight Separation and support it in our approximate solver described in Section 4.3.

Theorem 4.4 (WS upper bound)**.** *For any $\alpha, \beta \in (0,1)$ such that $\alpha < \beta$: there exists a solution to the Weight Separation problem with $T \leq \frac{(\alpha+\beta)(1-\alpha)}{\beta-\alpha} n$.*

Note that the numerator $(\alpha+\beta)(1-\alpha)$ is always smaller than 1 for $0 < \alpha < \beta < 1$.

4.3 Swiper: Approximate solver for Weight Reduction problems

System	number of tickets using Swiper						
	WR and WQ				WS		
	$\alpha_w = 1/4$ $\alpha_n = 1/3$ $\beta_w = 3/4$ $\beta_n = 2/3$	$\alpha_w = 1/3$ $\alpha_n = 3/8$ $\beta_w = 2/3$ $\beta_n = 5/8$	$\alpha_w = 1/3$ $\alpha_n = 1/2$ $\beta_w = 2/3$ $\beta_n = 1/2$	$\alpha_w = 2/3$ $\alpha_n = 3/4$ $\beta_w = 1/3$ $\beta_n = 1/4$	$\alpha = 1/4$ $\beta = 1/3$	$\alpha = 1/3$ $\beta = 1/2$	$\alpha = 2/3$ $\beta = 3/4$
Aptos [14, 15] $W = 8.47 \times 10^8 \quad n = 104$	85	235	27	110	385	98	437 (+1)
Tezos [64, 75] $W = 6.76 \times 10^8 \quad n = 382$	133	425	61 (+8)	258 (+1)	670	233 (+2)	811
Filecoin [62, 95] $W = 2.52 \times 10^{19} \quad n = 3\,700$	3 091	8 233	1 533	4 691	10 485	4 838	11 858
Algorand [11, 110] $W = 9.72 \times 10^9 \quad n = 42\,920$	745	13 475	293	6 258	46 009	2 188	64 189

Table 4.2: Number of tickets allocated by the Swiper protocol on sample weight distributions. In the few cases when the linear mode yields more tickets than the standard (full) mode, the difference is written in parentheses.

To provide a constructive proof for Theorems 4.1, 4.3 and 4.4 as well as to facilitate practical applications of weight reduction problems, we designed Swiper—a fast approximate solver for the three weight reduction problems defined in this chapter. Swiper enjoys a number of desirable properties:

1. **Robustness:** It always respects the upper bounds stated in Section 4.2. This means that even under a malicious distribution of weights, the number of assigned tickets will be within a known limit, linear in the number of parties. We present the algorithm in Section 4.3.1 and prove the upper bounds in Section 4.8.

2. **Determinism:** Swiper is a deterministic protocol. Hence, when the initial weights are common knowledge, each party can run it locally and all parties will obtain the same result. This eliminates the need for executing any complex protocol to agree on the ticket assignment.

3. **Allocation efficiency:** As we explore in detail in Section 7, Swiper performs remarkably well on real-world weight distributions, often allocating far fewer tickets than predicted by the upper bounds. In Table 4.2, we summarize the number of tickets allocated by Swiper on the distribution of funds in four major blockchain systems [11, 15, 62, 64] with some example thresholds. Notice that, in many cases, the number of tickets is actually below the number of users. This happens partly due to the distributions being significantly skewed and a large number of users actually owning only a small fraction of the total funds.

4. **Computational efficiency:** Assuming that the thresholds $(\alpha, \alpha_w, \alpha_n, \beta, \beta_w, \beta_n)$ are constants, the runtime of Swiper is either $\tilde{O}(n)$ (in `--linear` mode) or $\tilde{O}(n^2)$ (in standard mode). The difference in the implementation of the two modes is detailed in Section 4.3.1. Both modes respect the upper bounds and, as can be seen in Table 4.2, in practice, usually yield identical or almost identical results.

4.3.1 Algorithm and implementation

Overall structure. In Swiper, we consider ticket assignments of a special form. Let c be a fixed number between 0 and 1 (we will precisely specify c later in this section). Let $t(s, k)$ be the result of the following procedure: first, let $t_i := \lfloor sw_i + c \rfloor$; then, consider the parties that ended up "on the border", i.e., that would lose a ticket if we decreased s any further[1] and take 1 ticket from all but arbitrary (yet deterministically chosen) k of them.

More formally, let $\mathcal{B}_s := \{i \mid sw_i + c \text{ is integer}\}$ and $\mathcal{K}_{s,k} := \{\text{arbitrary } k \text{ members of } \mathcal{B}_s\}$. Then:

$$t(s,k)_i := \begin{cases} \lfloor sw_i + c \rfloor - 1, & \text{if } i \in (\mathcal{B}_s \setminus \mathcal{K}_{s,k}) \\ \lfloor sw_i + c \rfloor, & \text{otherwise} \end{cases}$$

The crucial observation is that, despite having two indices, this family of ticket assignments can be totally ordered, each ticket assignment having precisely one ticket more than the previous one (after removing duplicates). Indeed, let $T_{s,k} := \sum_{i=1}^{n} t(s,k)_i$. Then, for $0 < k < |\mathcal{B}_s|$, by definition, $T(s, k+1) = T(s,k) + 1$. Moreover, if s' is the smallest number greater than s such that $|\mathcal{B}_{s'}| \neq 0$, then $T(s', 1) = T(s', 0) + 1 = T(s, |\mathcal{B}_s|) + 1$. For any s'' in between s and s', $t(s'', *) = t(s', 0) = t(s, |\mathcal{B}_s|)$.

Swiper finds a *local minimum* in this family of ticket assignments, i.e., s^* and k^* such that $t(s^*, k^*)$ is viable (satisfies the problem requirements), but, for any sufficiently small ε and any k', $t(s^* - \varepsilon, k')$ is not viable and neither is $t(s^*, k^* - 1)$.

Theoretical foundations. In Section 4.8, we prove that, by selecting the constant c as α_w in case of Weight Restriction, $(1 - \beta_w)$ in case of Weight Qualification, and $\frac{\alpha+\beta}{2}$ in case of Weight Separation, the resulting ticket assignment always satisfies the bounds stated in Section 4.2 (Theorems 4.1, 4.3 and 4.4).[2] The proof works by demonstrating that any invalid ticket assignment of this form yields at least one fewer tickets than the stated upper bounds. Any local minimum yields just 1 ticket more than *some* invalid ticket assignment and, thus, *fewer or equal* to the upper bounds. This proof structure is important for achieving practical efficiency.

[1] This corresponds to all i such that $sw_i + c$ is an integer.

[2] To obtain these specific values, we first considered the general case for an arbitrary c and then found the values of c that minimized the upper bounds.

Bootstrapping the solution. As mentioned above, if the ticket assignment yielded by some tuple (s, k) is invalid (does not satisfy the problem's requirement), then the total number of tickets in this ticket assignment must be smaller than the upper bound. Conversely, if some tuple (s, k) yields a ticket assignment with the total number of tickets greater or equal to the upper bound, we can conclude that the resulting ticket assignment is valid. This fact alone allows us to quickly arrive at a valid solution satisfying the upper bound by simply finding a tuple (s, k) that yields the number of tickets exactly equal to the upper bound. This can be done efficiently with a binary search.

Finding the local minimum. Thanks to the fact that we are only looking for a *local* minimum in the considered family of ticket assignments, we can find it efficiently with a binary search, assuming an efficient algorithm for verifying the validity of a ticket assignment. However, *in the general case*, verifying the validity of a ticket assignment looks a lot like a (co-)NP-hard problem. Indeed, one can easily see that verifying a solution to Weight Restriction as defined in Section 4.2 is equivalent to solving a particular instance of Knapsack—the famous NP-hard optimization problem [89].

Fortunately, for the specific family of ticket assignments that Swiper considers (denoted as $t(s, k)$ earlier in this section), an efficient algorithm does exist. Indeed, we have already established that any ticket assignment in this family with the total number of tickets (T) exceeding the upper bound is valid. If, on the other hand, T is smaller than the upper bound, then we can use the "dynamic programming by profits" approach [89, Lemma 2.3.2] to solve Knapsack in time $O(Tn)$. Assuming αs and βs to be constant, T is $O(n)$ and $O(Tn) = O(n^2)$.

Practical efficiency and the `--linear` mode. Solving Knapsack to verify the validity of $t(s, k)$ is the main bottleneck for the algorithm. To achieve better practical efficiency, Swiper uses well-known quasilinear-time Knapsack lower and upper bounds to filter out as many solutions as possible without invoking the knapsack solver.

The upper bound allows us to implement a *conservative check*, i.e., it may yield false negatives (falsely declaring $t(s, k)$ as invalid), but never false positives (falsely declaring $t(s, k)$ as valid). In `--linear` mode, Swiper only relies on the upper bound and is guaranteed to find a valid solution, albeit not necessarily a locally minimal one.

Additionally, the lower bound allows us to implement a *liberal check*, i.e., it may yield false positives, but never false negatives. By combining the two, we can implement a quick test that can return one of the three values ("valid", "invalid", or "uncertain"). In the full mode (i.e., when `--linear` is not provided), Swiper only invokes the full knapsack solver (with $O(n^2)$ time complexity) when the quick test returns "uncertain", which speeds up the algorithm by a more than a factor of 3 on inputs with large enough resulting number of tickets.

Prototype implementation. We provide the full code for a prototype of Swiper and the data used to generate Table 4.2 in a public GitHub repository[3]. The prototype is implemented in Python, with JIT compilation used for certain computation-heavy parts. It utilizes the `Fraction` class to avoid any possible rounding errors. If sub-second latencies are required by the application, an implementation in a more performance-oriented programming language as well as the use of rounding (be it floating- or fixed-point arithmetic) may be necessary.

4.4 Applications of Weight Restriction

4.4.1 Distributed random number generation

As a motivating example for Weight Restriction, consider the *Distributed Random Number Generation* problem. Typically, it needs to satisfy two properties:

- If *all* honest parties cooperate, they can generate the next random number;
- Unless *at least one* honest party wants to open the next random number, it remains completely unpredictable to the adversary.

[3] `https://github.com/DCL-TelecomParis/swiper`

Perhaps, the simplest way it can be achieved [119] is by having a trusted party generate the random number and pre-distribute it using secret sharing [123], such that each party gets a number t_i of shares and any subset of parties possessing at least $\lceil \alpha_n T \rceil$ shares (where $T = \sum_{i=1}^{n} t_i$) can reconstruct the secret, but no set of parties possessing less than this amount of shares can learn anything about the secret.

Thus, by setting α_w to the resilience of the protocol ($\alpha_w := f_w$) and $\alpha_n \leq \frac{1}{2}$, we can guarantee that:

• Honest participants will receive more than $(1 - \alpha_n)T \geq \lceil \alpha_n T \rceil$ shares and, hence, will be able to reconstruct the random number.

• Corrupt participants will receive less than $\alpha_n T$ shares and, hence, will not be able to reconstruct the random number unless some honest party also wants to open it;

Practical randomness beacons [37, 121] operate similarly, only employing *unique threshold signatures* [24, 124] in order to be able to reuse the same secret multiple times. The described weighted solution still applies to such approaches unchanged.

4.4.2 Blunt Secret Sharing and derivatives

In cryptography, certain actions have an associated access structure $\mathbb{A}$ that determines all sets of parties that are able to perform these actions once they collaborate. Traditional $(n, k+1)$-threshold systems can be seen as a particular access structure $\mathbb{A}_n(\alpha) = \{P \subseteq [n] : |P| > \alpha n\}$, where $\alpha := \frac{k}{n}$. Analogously, a *weighted* threshold access structure can be defined as $\mathbb{A}_w(\alpha) = \{P \subseteq \Pi : \sum_{i \in P} w_i > \alpha \sum_{i \in \Pi} w_i\}$.

We can also define the *adversary structure* $\mathbb{F} \subseteq 2^{\Pi}$, the set of all sets of parties that can be simultaneously corrupted at any given execution. Often, the adversary structure is also defined by a threshold, with a maximum corruptible weight fraction f_w, i.e., $\mathbb{F}_w(f_w) = \{P \subset \Pi : \sum_{i \in P} w_i < f_w \sum_{i \in \Pi} w_i\}$.

While threshold access structures are commonly studied in cryptography and are applied in numerous distributed protocols, in practice, as we illustrate in Section 4.6, it is often sufficient if the access structure provides the following two properties, generalizing the requirements of the random beacon presented in Section 4.4.1:

• There exists at least one set entirely composed of honest parties that belongs to the access structure. This typically guarantees the accompanying protocol's *liveness properties*.

• Any set containing only corrupt parties does not belong to the access structure, as this would break *safety properties*.

Hence, we define a *blunt access structure* as follows:

Definition 4.5 (Blunt access structure)**.** *Given a set of parties* Π *and the adversary structure* $\mathbb{F} \subseteq 2^{\Pi}$*,* $\mathbb{A}$ *is a blunt access structure w.r.t.* $\mathbb{F}$ *if* $(\forall F \in \mathbb{F} : F \notin \mathbb{A})$ *and* $(\exists A \in \mathbb{A} : A \cap F = \varnothing)$*.*

The following theorem shows that solving WR is sufficient to implement weighted cryptographic protocols with blunt access structures by a reduction to their nominal counterparts.

Theorem 4.6. *Given a set of parties, a protocol* $\mathcal{P}$ *implementing a cryptographic primitive with nominal threshold access structure* $\mathbb{A}_n(\alpha_n)$*, for* $\alpha_n \leq \frac{1}{2}$*, we obtain a protocol* $\mathcal{P}'$ *implementing a blunt access structure w.r.t. adversarial structure* $\mathbb{F}_w(f_w)$*, assuming* $f_w < \alpha_n$*, by solving Weight Restriction with the corresponding parameters* α_n *and* $\alpha_w := f_w$*. This is accomplished by instantiating* $\mathcal{P}$ *with* $\hat{n} = T$ *virtual users and allowing party* i *to control* t_i *of them.*[4]

Proof. By definition of WR, once it distributes T tickets, the number of tickets (and, hence, virtual users) allocated to the corrupt parties will be less than $\alpha_n T$. Hence, no element of the adversary structure shall appear in the resulting access structure. In addition, honest participants will receive more than $(1 - \alpha_n)T \geq \alpha_n T$ (recall that $\alpha_n \leq \frac{1}{2}$) tickets (and, hence, virtual users), ensuring that there exists a set consisting of only honest parties in the access structure. □

[4]Recall that t_i is the number of tickets assigned to party i and T is the total number of tickets assigned by the solution to the weight reduction problem (in this case, to WR). See Section 4.2 for details.

Note that all participants must agree on how many virtual users are assigned to each party, as nominal protocols typically assume that the membership is common knowledge. To this end, it is sufficient for all parties to run an agreed upon *deterministic* weight-restriction protocol (e.g., Swiper).

Among other things, this way, one can obtain weighted versions of secret sharing [123], distributed random number generation [37], threshold signatures [24], threshold encryption [54], and threshold fully-homomorphic encryption [85], all with blunt access structures. In the next section, we discuss how to do it for other access structures.

4.4.3 Tight Secret Sharing and derivatives

Although a blunt access structure is sufficient for a large spectrum of applications, more restrictive access structures are sometimes necessary as well. Here, we present a straightforward approach that involves just one extra round of communication to transform a blunt access structure into a weighted threshold access structure.[5] This means that our construction can be readily utilized in any protocol that already uses threshold cryptography without requiring significant redesign efforts.

Given a protocol $\mathcal{P}$ implementing a certain primitive of distributed cryptography (e.g., threshold signatures [54]) with a blunt access structure, we can obtain a protocol $\mathcal{P}'$ implementing the same protocol with a weighted threshold access structure $\mathbb{A}_w(\beta)$ as follows: whenever an honest party wants to perform an action $\mathcal{A}$ (e.g., produce a threshold signature), instead it simply broadcasts a message "voting" for the action to be performed, without actually revealing any secret data (e.g., its threshold signature share). Then, when an honest party receives such votes from parties with a total weight more than βW, it participates in the action $\mathcal{A}$, according to the underlying protocol $\mathcal{P}$ (e.g., broadcasts its threshold signature share). Thus, we can notice that:

1. Unless a threshold of parties (potentially including Byzantine) cast votes for $\mathcal{A}$, no honest party will participate in $\mathcal{A}$ in $\mathcal{P}$. Thus, by Theorem 4.5, action $\mathcal{A}$ will not be performed;

2. If a threshold of parties cast votes for $\mathcal{A}$, all honest parties will eventually participate in $\mathcal{A}$ according to $\mathcal{P}$, thus, by Theorem 4.5, the action will be performed.

4.4.4 Black-Box transformation

The same approach of allocating a number of virtual users according to the number of tickets as described in Section 4.4.2 can be applied to arbitrary distributed protocols.

Given a nominal protocol $\mathcal{P}$, the "virtual users" approach allows us to define a protocol $\mathcal{P}'$ that operates in the weighted model by, essentially, emulating the nominal model, as long as we can solve Weight Restriction with parameters $\alpha_w := f_w$ and $\alpha_n := f_n$. If $f_w < f_n$, by Theorem 4.1, $T = \sum_{i \in [n]} t_i$ will be at most $O\left(\frac{n}{f_n - f_w}\right)$. In $\mathcal{P}'$, each party i participates in $\mathcal{P}$ with t_i virtual identities. Two components of the transformation depend on the problem at hand (but not on the underlying protocol $\mathcal{P}$):

1. Mapping the input of i in $\mathcal{P}'$ to the inputs of its virtual identities in $\mathcal{P}$;

2. Treatment of the outputs of i's virtual identities in $\mathcal{P}$ to produce the outputs in $\mathcal{P}'$.

We illustrate the black-box transformation with two examples: Validated Byzantine Agreement [36] and Single Secret Leader Election [26].

Consensus. For concreteness, let us consider the problem of Validated Byzantine Agreement (VBA) [36]. However, one can easily verify that the same logic will apply to most, if not all, of the many types of consensus and state machine replication, including both crash and Byzantine fault-tolerant ones.

Definition 4.7. *A protocol solves validated Byzantine agreement with external validity predicate $\mathcal{V}$ if it satisfies the following conditions:*

Liveness: *Each honest party outputs a value.*

[5] In fact, this can be further generalized to arbitrary access structures.

Agreement: *No two honest parties can output different values.*

External Validity: *If an honest party outputs v, then $\mathcal{V}(v)$ holds.*

Integrity: *If all parties are honest, and if some party decides v, then v is the input of some party.*

Efficiency: *The communication complexity is probabilistically uniformly bounded.*

Consider an arbitrary protocol $\mathcal{P}$ that solves the problem for some external validity predicate $\mathcal{V}$. Let $\mathcal{P}'$ be the protocol obtained from $\mathcal{P}$ by applying the transformation described above with the problem-specific part defined as follows:

1. The input of all virtual identities of party i in $\mathcal{P}$ is the same as i's input in $\mathcal{P}'$;

2. If $t_i \neq 0$, party i outputs the value output by its first virtual identity and sends it to all parties j such that $t_j = 0$. If $t_i = 0$, it waits for messages from parties with total weight greater than $f_w W$ vouching for the same output v and outputs v.

By construction and the definition of WR, assuming that at most a fraction f_w of the total weight is corrupted, at most a fraction f_n of virtual identities will be corrupted and, hence, assuming $\mathcal{P}$ solves VBA with nominal resilience f_n, the simulated protocol will satisfy the properties of VBA. One can easily verify that each of the five properties will be satisfied for $\mathcal{P}'$ as well. Notice, in particular, that efficiency will still be satisfied as the total communication complexity will be increased by only a constant factor (assuming f_w and f_n to be constants).

Single Secret Leader Election. SSLE [26] is a distributed protocol that has as an objective to select one of the participants to be a leader with an additional constraint that only the elected party knows the result of the election. Then, once the leader is ready to make a proposal, it reveals itself and other participants can then correctly verify that the claiming leader was indeed elected by the protocol.

The original paper [26] contains nominal solutions for the protocol relying on ThFHE [27] and on shuffling a list of commitments under the DDH assumption. The authors initially suggest that their protocols could support weights by replicating each party to match their weights. This approach is identical to the transformation described in this section with the exception that it does not include weight reduction and, thus, exhibits overhead proportional to the total weight (which can be prohibitively large, see Table 4.2). We can solve this issue by applying Weight Restriction at the cost of lowering the resilience by an arbitrarily small constant ϵ ($f_w = f_n - \epsilon$).

However, the original problem definition requires the election to be *fair*, that is, for the probability of each party being elected to be uniform. It is easy to see that, as a result of applying weight reduction, this property will not be maintained. Instead, we can relax it to an alternative property of *chain-quality*, requiring that the fraction of blocks produced by corrupt parties should not surpass a constant fraction α when the adversary might control a fraction of the weights up to f_w. Our transformation then trivially solves this problem for $\alpha := f_n$.

Properties such as *fairness* are one of the limitations of our transformations since any property that is a function of the weight of the parties may not be preserved after the transformation is applied. We discuss fairness in slightly more detail and speculate about possible fixes to this issue in Section 4.12.

4.5 Applications of Weight Qualification

4.5.1 Erasure-Coded Storage and Broadcast

Erasure-coded storage systems [38, 80, 116, 120, 141], also known under the names of Information Dispersal Algorithms (IDA) [120] and Asynchronous Verifiable Information Dispersal (AVID) [38], are crucial to many systems for space and communication-efficient, secure, and fault-tolerant storage. Moreover, as demonstrated in [38], they can yield highly communication-efficient solutions to the very important problem of asynchronous Byzantine Reliable Broadcast [29, 34], a fundamental building block in distributed computing that, among other things, serves as the basis for many practical consensus [50, 57, 88, 111, 128], distributed key generation [3, 53], and mempool [50] protocols.

The challenge of applying these protocols in the weighted setting is that (k, m) erasure coding, by definition, converts the original data into m discrete *fragments* such that any k of them are sufficient

to reconstruct the original information. Thus, each party will inevitably get to store an integer number of these fragments, and the smaller m is, the more efficient the encoding and reconstruction will be. Moreover, for the most commonly used codes–Reed Solomon–the original message must be of size at least $k \log m$ bits. Hence, using a large m may lead to increased communication as the message may have to be padded to reach this minimum size. As we illustrate in this section, determining the smallest "safe" number of fragments to give to each party is exactly the Weight Qualification problem defined in Section 4.2.

Let us consider the example of [38] as it is the first erasure-coded storage protocol tolerating Byzantine faults. We believe Weight Qualification can be applied analogously to other similar works.

This protocol operates in a model where any t out of n parties can be malicious or faulty, where $t < \frac{n}{3}$. In other words, it has the nominal fault threshold of $f_n = \frac{1}{3}$. The protocol encodes the data using $(t+1, n)$ erasure coding, and the data is considered to be reliably stored once at least $2t+1$ parties claim to have stored their respective fragments. The idea is that, even if t of them are faulty, the remaining $t+1$ parties will be able to cooperate to recover the data.

In order to make a weighted version of this protocol, instead of waiting for confirmations from $2t+1$ parties, one needs to wait for confirmations from a set of parties that together possess more than a fraction $2f_w$ of total weight, where $f_w = f_n = \frac{1}{3}$. A subset of weight less than f_w of these parties may be faulty. Hence, for the protocol to work, it is sufficient to guarantee that any subset of total weight more than $2f_w - f_w = f_w$ gets enough fragments to reconstruct the data. To this end, we can apply the WQ problem with the threshold $\beta_w = f_w$. We can set β_n to be an arbitrary number such that $0 < \beta_n < \beta_w$. Then, we can use $(\lceil \beta_n T \rceil, T)$ erasure coding, where T is the total number of tickets allocated by the WQ solution. Hence, whenever a set of parties of weight more than $2f_w$ claim to have stored their fragments, we will be able to reconstruct the data with the help of the correct participants in this set. As for the rest of the protocol, it can be converted to the weighted model simply by applying weighted voting, as was discussed in Section 4.1.2.

As a result, we manage to obtain a weighted protocol for erasure-coded verifiable storage with the same resilience as in the nominal protocol ($f_w = f_n = \frac{1}{3}$). The "price" we pay is using erasure coding with a smaller rate (β_n instead of f_w), i.e., storing data with a slightly increased level of redundancy. However, note that β_n can be set arbitrarily close to f_w, at the cost of more total tickets and, hence, more computation.

Example instantiations

The communication and storage complexity of these protocols depends linearly on the rate of the erasure code. Using Reed-Solomon with Berlekamp-Massey decoding algorithm, the decoding computation complexity [70] is $O(m^2 \cdot \frac{M}{rm}) = O(\frac{m}{r} \cdot M)$, where M is the size of the message (which we do not affect), r is the rate of the code (in our case, $r = \beta_n$), and m is the number of fragments (in our case, the number of tickets allocated by the solution to the WQ problem). For the sake of illustration, let us fix β_n to be $\frac{1}{4}$. Then, the rate of the code used in the weighted solution will be $\frac{4}{3}$ times smaller than in the nominal solution. For the number of fragments m, let us substitute the upper bound from Theorem 4.3 ($m \leq \left\lceil \frac{\beta_w(1-\beta_w)}{\beta_w - \beta_n} n \right\rceil$). For $\beta_w = \frac{1}{3}$ and $\beta_n = \frac{1}{4}$, $m \leq \frac{8}{3}n$. Hence, the overall slow-down compared to the nominal solution is $\frac{8}{3} \cdot \frac{4}{3} \approx 3.56$.

One can also consider using FFT-based decoding algorithms [87]. Since the complexity of the FFT-based decoding depends only polylogarithmically on the number of fragments m, one can select the rate of the code ($r = \beta_n$) to be much closer to β_w and, thus, minimize communication and storage overhead.

Some protocols [115] are designed for higher reconstruction thresholds, which allows them to be more communication- and storage-efficient compared to [38]. For these cases, we will need to set $\beta_w := \frac{2}{3}$. By setting $\beta_n := \frac{1}{2}$ and applying the upper bound from Theorem 4.3, we will obtain the same reduction of factor $\frac{4}{3}$ in rate and 2 times fewer tickets: $m \leq \frac{1/3 \cdot 2/3}{2/3 - 1/2} n = \frac{4}{3}$. The computational overhead will be $\frac{4}{3} \cdot \frac{4}{3} \approx 1.78$.

4.5.2 Error-Corrected Broadcast

The exciting work of [52] illustrated how one can avoid the need for complicated cryptographic proofs in the construction of communication-efficient broadcast protocols by employing error-correcting codes, thus

enabling a better communication complexity when a trusted setup is not available. The protocol of [52] can be used for the construction of communication-efficient Asynchronous Distributed Key Generation [3, 53] protocols.

Similarly to erasure codes, error-correcting codes convert the data into m discrete fragments, such that any k of them are sufficient to reconstruct the original information. However, they have the additional property that the data can be reconstructed even when some of the fragments input to the decoding procedure are invalid or corrupted. Reed-Solomon decoding allows correcting up to e errors when given $k + 2e$ fragments as input.

The protocol of [52] tolerates up to t failures in a system of $n \geq 3t + 1$ parties (for simplicity, we will consider the case $n = 3t + 1$). Its key contribution is the idea of *online error correction*. Put simply, the protocol first ensures that:

- Every honest party obtains a cryptographic hash of the data to be reconstructed;
- Every honest party obtains its chunk of the data.

Then, in order to reconstruct a message, an honest party solicits fragments from all other parties and repeatedly tries to reconstruct the original data using the Reed-Solomon decoding and verifies the hash of the output of the decoder against the expected value. As the protocol uses $k = t + 1$ and $m = n$, after hearing from all $2t + 1$ honest and $e \leq t$ malicious parties, it will be possible to reconstruct the original data (as $2t + 1 + e \geq k + 2e$, for $k = t + 1$).

To convert this protocol into the weighted model, it is sufficient to make sure that all honest parties together possess enough fragments to correct all errors introduced by the corrupted parties. To this end, we will apply the WQ problem. We will set β_w to the fraction of weight owned by honest parties, i.e., $\beta_w := 1 - f_w = \frac{2}{3}$ (where f_w will be the resilience of the resulting weighted protocol, $f_w = f_n = \frac{1}{3}$). However, it is not immediately obvious how to set β_n to allow the above-mentioned property.

If we want to use error-correcting codes with rate r, we need to guarantee that the fraction of fragments received by the honest parties (which is at least β_n) is at least $r + e$, where e is the fraction of fragments received by the corrupted parties. However, since honest parties get at least the fraction β_n of all fragments, then $e \leq 1 - \beta_n$. Hence, we need to set β_n so that $\beta_n \geq r + (1 - \beta_n)$. We can simply set $\beta_n := \frac{r}{2} + \frac{1}{2}$ for arbitrary $r < \frac{1}{3}$.

Example instantiation

For the sake of an example, we can set $\beta_w := \frac{2}{3}$, $r := \frac{1}{4}$ and $\beta_n := \frac{5}{8}$. Then, using the bound from Theorem 4.3, the number of tickets will be at most $\frac{2/3 \cdot 1/3}{2/3 - 5/8} \cdot n \leq \frac{16}{3} n$.

As was discussed above, for erasure codes, we can either use the Berlekamp-Massey decoding algorithm or the FFT-based approaches. The same applies to error-correcting codes. As most practical implementations use the former, we will focus on it. In this case, the communication overhead will be $\frac{r_n}{r_w}$, where $r_n = \frac{1}{3}$ is the rate used in the nominal protocol and r_w is the rate used for the weighted protocol (in the example above, $r = \frac{1}{4}$). The computation overhead is $\frac{r_n}{r_w} \cdot \frac{T}{n}$, where T is the number of tickets allocated by the WQ solution (in the example above, $T \leq \frac{16}{3} n$ in the worst case). Hence, for the example parameters, the worst-case computational overhead is $\frac{4}{3} \cdot \frac{16}{3} \approx 7.11$.

4.6 Derived Applications

In this section, we discuss indirect applications of weight reduction problems that are obtained by using one or multiple building blocks discussed in Sections 4.4 and 4.5. For all applications discussed here, we manage to avoid losing resilience despite applying weight reduction. In all cases, the majority of the protocol logic should be converted to the weighted model by applying weighted voting, as discussed in Section 4.1.2.

4.6.1 Asynchronous State Machine Replication

For asynchronous state machine replication protocols [50, 57, 88, 111, 128], we simply need to use a weighted communication-efficient broadcast protocol (discussed in Section 4.5) and weighted distributed

random number generation (discussed in Section 4.4.1). distributed number generation part can use a nominal protocol with threshold $\alpha_n = \frac{1}{2}$ and set $\alpha_w := \frac{1}{3}$, which is the resilience of the rest of the protocol. Thus, in some sense, we level the resilience of different parts of the protocol, without affecting the resilience of the composition.

4.6.2 Validated Asynchronous Byzantine Agreement

The same approach can be applied to generate randomness for Validated Asynchronous Byzantine Agreement (VABA) [4, 36].

These protocols also require tight threshold signatures. However, in practice, multi-signatures [24, 117] are usually applied instead as they have almost no overhead over threshold signatures on the system sizes where such protocols could be applied (below 1000 participants): it suffices to append the multi-signature with an array of n bits, indicating the set of parties that produced the signature. Then, along with the verification of the validity of the multi-signature itself, anyone can verify that the signers together hold sufficient weight.

Alternatively, one could apply the approach described in Section 4.4.3 to implement tight weighted threshold signatures. However, it would lead to an increase in message complexity of the resulting protocol, which we want to avoid.

Finally, an ad-hoc weighted threshold signature scheme can be applied, such as the one recently proposed in [51]. Note that these signatures cannot be used for distributed randomness generation as they lack the necessary uniqueness property, and thus we still need to apply Swiper to obtain a complete protocol.

4.6.3 Consensus with Checkpoints

We can apply the same approach for checkpointing proof-of-stake consensus protocols [18], but this time for blunt threshold signatures (as discussed in Section 4.4.2) instead of random number generation. If, for some reason, one wants to use a tight threshold signature, the approach described in Section 4.4.3 can be applied at the cost of just 1 additional message delay per checkpoint.

Compared to ad-hoc solutions for weighted threshold signatures [51], we claim that our approach is more computationally efficient as it is basically as fast as the underlying nominal protocol. For example, 2 pairings to verify a BLS signature [24] compared to 13 pairings to verify a signature in [51]. Moreover, the weight reduction approach is more general and can support other types of threshold signatures, such as RSA [124] and Schnorr [129], the latter being particularly important in the context of checkpointing to Bitcoin [18].

4.7 Analyzing Weight Restriction on sample systems

Data sets. We analyzed our protocol using four real-world data sets for weight distribution: Aptos [14, 15], Tezos [64, 75], Filecoin [62, 95], and Algorand [11, 110]. For the reader's convenience, we provide the results for all the datasets in a separate section 4.10 and present the results for only one blockchain (Tezos) in Figure 1 as an example.

Experiment description. We performed two kinds of experiments on real blockchain data. In the first experiment, shown in the left column of Figure 1, we analyzed the influence of the choice of parameters α_w and α_n for the original data retrieved from the blockchains; the value of α_n was varied in the range $[0.1, 1]$, while the value of α_w was tested in the range $[0.1 \times \alpha_n, 0.9 \times \alpha_n]$. In the experiments showcased in the right column of Figure 1, we kept these parameters fixed and analyzed the influence of the number of parties in the metrics we tracked. In order to simulate having the same blockchain with different numbers of parties, we used the statistical technique known as bootstrapping. To this end, we performed 100 experiments sampling parties with replacement from the blockchain data and taking the average of the results.

In each experiment, we tracked the total number of tickets distributed, the maximum number of tickets held by a single party, and the number of parties that get at least one ticket (in the figures, we label them as the number of holders). In Figure 1, we show the results for the Tezos blockchain. The results for Algorand, Aptos, and Filecoin are available in Section 4.10. The analysis of the results

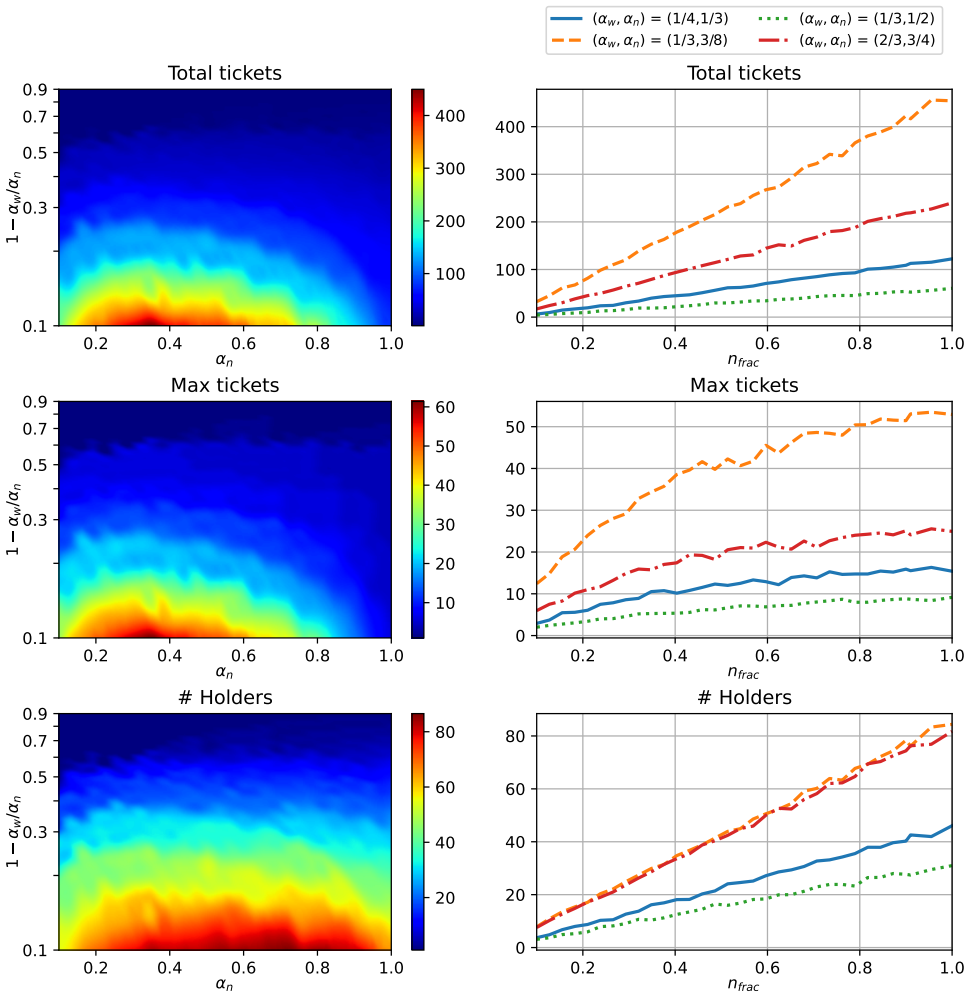

Figure 1: Experiment results using Tezos (100 samples per data point)

reveals the following information: the upper bound given in Section 4.2 is very pessimistic for weight distributions emerging in practice, with the total number of tickets rarely surpassing the number of parties for different values of α_n and α_w. The total number of tickets varies extremely close to a linear function on the number of parties, as well as the number of holders. The maximum number of tickets, on the other hand, seems to saturate when the number of parties in absolute terms surpasses the order of magnitude of 1000, remaining almost constant after that point.

4.8 Proofs

In this section, we provide formal proofs for Theorems 4.1, 4.3 and 4.4.

4.8.1 Upper bounds on Weight Restriction and Weight Separation

Let us start with some auxiliary definitions. A *ticket assignment* t is a vector of n numbers: $t_1, \ldots, t_n \in \mathbb{Z}_{\geq 0}$. With a slight abuse of notation, for a ticket assignment t and a set $S \subseteq [n]$, we use notation $t(S)$ to denote $\sum_{i \in S} t_i$. Let us say that a ticket assignment t is *viable* if $t([n]) \neq 0$ and $\forall S \subseteq [n]$: if $w(S) < \alpha_w W$, then $t(S) < \alpha_n t([n])$, that is if it satisfies the requirements of the Weight Restriction problem as defined in Section 4.2.

In this section, we formally prove Theorem 4.1 by constructing a viable ticket assignment $\hat{t}$ such that $\hat{t}([n]) \leq \left\lceil \frac{\alpha_w(1-\alpha_w)}{\alpha_n - \alpha_w} n \right\rceil$. As the starting point, we consider a family of ticket assignments parameterized by a single number $s > 0$:

$$(t_s)_i := \lfloor w_i s + \alpha_w \rfloor.$$

Let s^* be a *locally minimal* viable value for s, i.e., a positive number such that t_{s^*} is viable, but $t_{s^*-\varepsilon}$ is not, for any sufficiently small ε. Since we already proved that viable values of s exist, it is easy to see that such s^* exists. Moreover, there must be some j such that $s^* w_j + \alpha_w$ is an integer. Indeed, if this does not hold, we would be able to slightly decrease s^* without changing the ticket assignment, which would contradict the assumption that s^* is a local minimum. Let $t^* := t_{s^*}$ and $J := \{j \in [n] \mid s^* w_i + \alpha_w \text{ is an integer}\}$. Let t' be a ticket assignment in which we take one ticket from each party in J, i.e.:

$$t'_i := \begin{cases} t^*_i - 1 & \text{if } i \in J \\ t^*_i & \text{otherwise} \end{cases}$$

Notice that t' is equal to $t_{s^*-\varepsilon}$ for a sufficiently small $\varepsilon > 0$.[6] Hence, by construction, t' is not viable. Now, let us consider a set of "intermediate" ticket assignments: we will be taking tickets from parties in J as long as the ticket assignment stays viable. We will end up with two ticket assignments: $\hat{t}$ and $\hat{\hat{t}}$ such that $\hat{t}$ is viable and $\hat{\hat{t}}$ is not, and $\hat{t}([n]) = \hat{\hat{t}}([n]) + 1$. All that is left is to prove that $\hat{t}([n]) \leq \left\lceil \frac{\alpha_w(1-\alpha_w)}{\alpha_n - \alpha_w} n \right\rceil$ or, equivalently, that $\hat{\hat{t}}([n]) \leq \left\lceil \frac{\alpha_w(1-\alpha_w)}{\alpha_n - \alpha_w} n \right\rceil - 1$.

Since $\hat{\hat{t}}$ is not viable, either $\hat{\hat{t}}([n]) = 0$ or there must exist a set $S \subseteq [n]$ such that $w(S) < \alpha_w W$ and $\hat{\hat{t}}(S) \geq \alpha_n \hat{\hat{t}}([n])$. As the former case is trivial, we will focus on the latter. Let us provide an upper bound on $\hat{\hat{t}}(S)$ and a lower bound on $\hat{\hat{t}}(\overline{S})$, where $\overline{S} := [n] \setminus S$. To this end, let us note that, for any $i \in [n]$, it holds that $\hat{\hat{t}}_i \geq w_i s^* + \alpha_w - 1$. Indeed, there are two cases to consider:

1. if $\hat{\hat{t}}_i = t^*_i$, the inequality holds trivially as $\hat{\hat{t}}_i = t^*_i = \lfloor w_i s^* + \alpha_w \rfloor$;

2. otherwise, $\hat{\hat{t}}_i = t^*_i - 1$. However, by construction, it means that $w_i s^* + \alpha_w$ is an integer and, thus $t^*_i = w_i s^* + \alpha_w$ and $\hat{\hat{t}}_i = w_i s^* + \alpha_w - 1$.

[6] Indeed, if we decrease s^* by any positive amount, each party in J will lose at least one ticket as they will step over the rounding threshold. However, it is also easy to see that ε can be made small enough so that no other party will lose a ticket and no party in J will lose more than one ticket.

Hence:

$$\begin{aligned}
\hat{\hat{t}}(S) &= \sum_{i \in S} \hat{\hat{t}}_i \\
&\leq \sum_{i \in S} t_i^* = \sum_{i \in S} \lfloor w_i s^* + \alpha_w \rfloor \\
&< \alpha_w W s^* + \alpha_w |S| \\
\hat{\hat{t}}(\overline{S}) &= \sum_{i \notin S} \hat{\hat{t}}_i \\
&\geq \sum_{i \notin S} (w_i s^* + \alpha_w - 1) \\
&> (1 - \alpha_w) W s^* - (1 - \alpha_w)(n - |S|)
\end{aligned}$$

By construction, $\hat{\hat{t}}(S) \geq \alpha_n \hat{\hat{t}}([n])$ and $\hat{\hat{t}}([n]) = \hat{\hat{t}}(S) + \hat{\hat{t}}(\overline{S})$. Hence, $(1 - \alpha_n)\hat{\hat{t}}(S) \geq \alpha_n \hat{\hat{t}}(\overline{S})$. From this, we can derive an upper bound on s^*:

$$\begin{aligned}
&(1 - \alpha_n)\hat{\hat{t}}(S) \geq \alpha_n \hat{\hat{t}}(\overline{S}) \Rightarrow \\
\Rightarrow\ &(1 - \alpha_n)(\alpha_w W s^* + \alpha_w |S|) \\
&\quad > \alpha_n((1 - \alpha_w) W s^* - (1 - \alpha_w)(n - |S|)) \\
\Rightarrow\ &s^* < \frac{\alpha_n (1 - \alpha_w) n}{(\alpha_n - \alpha_w) W} - \frac{|S|}{W}
\end{aligned}$$

Finally, we can combine everything into an upper bound on $\hat{\hat{t}}([n])$:

$$\begin{aligned}
\hat{\hat{t}}([n]) &\leq \frac{\hat{\hat{t}}(S)}{\alpha_n} \\
&< \frac{\alpha_w}{\alpha_n}(W s^* + |S|) \\
&< \frac{\alpha_w}{\alpha_n}\left(\frac{\alpha_n (1 - \alpha_w) n}{\alpha_n - \alpha_w} - |S| + |S|\right) \\
&= \frac{\alpha_w (1 - \alpha_w)}{\alpha_n - \alpha_w} n
\end{aligned}$$

Since $\hat{\hat{t}}([n])$ is an integer and the inequality is strict, we can rewrite it as $\hat{\hat{t}}([n]) \leq \left\lceil \frac{\alpha_w (1 - \alpha_w)}{\alpha_n - \alpha_w} n \right\rceil - 1$. As, by construction, $\hat{t}$ is viable and $\hat{t}([n]) = \hat{\hat{t}}([n]) + 1$, we found a viable ticket assignment with at most $\left\lceil \frac{\alpha_w (1 - \alpha_w)}{\alpha_n - \alpha_w} n \right\rceil$ tickets, thus concluding the proof of Theorems 4.1 and 4.3. □

4.8.2 Upper bound on Weight Separation

Let $\gamma := \frac{\alpha + \beta}{2}$. For Weight Separation, we analyze a family of ticket assignments of form $t_{s,i} := \lfloor w_i s + \gamma \rfloor$. Let us consider the case when the WS conditions are violated, i.e., there exist sets S_1 and S_2 such that $w(S_1) < \alpha W$, $w(S_2) > \beta W$, and $t(S_1) \geq t(S_2)$. This means that at least one of two events happened: $t(S_1) \geq \gamma T$ or $t(S_2) < \gamma T$, or, equivalently, $t(\overline{S_2}) > (1 - \gamma)T$.

Let us first consider the case when $t(S_1) \geq \gamma T$. This can only happen when $s < \frac{\gamma(1 - \gamma)n}{(\gamma - \alpha)W}$. The proof

is done using the same set of techniques as in Section 4.8.1:

$$\begin{aligned} t(S_1) &= \sum_{i \in S_1} t_i = \sum_{i \in S_1} \lfloor w_i s + \gamma \rfloor \\ &\leq \sum_{i \in S_1} (w_i s + \gamma) \\ &< \alpha W s + \gamma |S_1| \\ t(\overline{S_1}) &= \sum_{i \notin S_1} \lfloor w_i s + \gamma \rfloor \\ &\geq \sum_{i \notin S_1} (w_i s + \gamma - 1) \\ &> \beta W s - (1-\gamma)(n - |S_1|) \end{aligned}$$

$$\begin{aligned} & t(S_1) \geq \gamma T \\ \Leftrightarrow\ & (1-\gamma) t(S_1) \geq \gamma t(\overline{S_1}) \\ \Rightarrow\ & (1-\gamma)(\alpha W s + \gamma |S_1|) > \gamma(\beta W s - (1-\gamma)(n - |S_1|)) \\ \Rightarrow\ & s < \frac{\gamma(1-\gamma)n}{(\gamma - \alpha)W} \end{aligned}$$

Analogously, in the case when $t(\overline{S_2}) > (1-\gamma)T$, we can prove (by substitution of $(1-\gamma)$ in place of γ and $(1-\beta)$ in place of α) that:

$$s < \frac{(1-\gamma)(1-(1-\gamma))n}{((1-\gamma)-(1-\beta))W} = \frac{\gamma(1-\gamma)n}{(\beta - \gamma)W}$$

We specifically chose $\gamma = \frac{\alpha+\beta}{2}$ so that the two bounds coincide: $s < \frac{2\gamma(1-\gamma)n}{(\beta-\alpha)W}$. Hence, it is sufficient to select $s := \frac{\gamma(2-\alpha-\beta)n}{(\beta-\alpha)W}$ to guarantee that neither of the two events happens and $t(S_1) < \gamma T \leq t(S_2)$.

Let us now compute a bound on the total number of tickets:

$$T \leq sW + \gamma n = \frac{(\alpha+\beta)(1-\alpha)}{\beta - \alpha} n$$

□

4.9 Exact solution using MIP

The way we formulate WR in section 4.2.1 can be directly translated into an instance of bi-level optimization problem [49]. In such problems, we define an *upper level* optimization problem which contains another (lower-level) optimization problem in its constraints, namely:

$$\begin{aligned} &\text{minimize} \sum_{i=1}^{n} t_i \\ &\text{subject to} \sum_{i=1}^{n} x_i t_i < \alpha_n \sum_{i=1}^{n} t_i \\ &\qquad \text{maximize} \sum_{i=1}^{n} x_i t_i \\ &\qquad \text{subject to} \sum_{i=1}^{n} w_i x_i < \alpha_w \sum_{i=1}^{n} w_i \\ &\qquad \sum_{i=1}^{n} t_i \geq 1 \\ &\qquad x_i \in \{0,1\}, t_i \in \{0,1,2,\dots\} \end{aligned}$$

Noticing that the inner optimization problem is the Knapsack problem, we can hard-code a dynamic programming by profits solution to the Knapsack problem into the constraints. Unfortunately, the resulting MIP has a lot, albeit a polynomial number, of constraints and, thus, is prohibitively slow for inputs of size larger than a couple of dozens.

4.10 Experiment Results

Figures 3 to 5 demonstrate the results of the experiments on the data from the stake distribution of 4 major blockchain systems: Aptos [14, 15], Tezos [64, 75], Filecoin [62, 95], and Algorand [11, 110]. The analysis of the experimental results is presented in Section 4.7.

4.11 Related Work

Knapsack. The Knapsack problem and its variations hold huge importance in theoretical computer science and have numerous applications in both theory and practice. The weight reduction problems studied in this chapter seem to be related to, or can even be seen as a variation of the famous Knapsack problem. For example, one can see Weight Restriction as the problem of constructing "worst possible" profits for a Knapsack instance given the weights and the capacity. We refer to [89] for a comprehensive survey on the topic.

Virtual users. The simplest solution for creating a weighted threshold cryptographic system is to simply have a user of weight w become w virtual users and to give one key to each of them. Shamir's paper describing his secret sharing scheme [123] puts forward this solution. However, in practice, the total weight tends to be prohibitively large, and "quantizing" it requires solving weight reduction problems, which is the main subject of this chapter.

Weighted voting. In [72], Gifford presents the idea of weighted voting for distributed storage systems. The paper suggests assigning weights to replicas according to the estimated failure probabilities and using weight-based quorums to store and retrieve data. We discuss the merits and limitations of this approach in Section 4.1.2. The goal of this chapter is to complement the weighted voting approach and design a framework for implementing weighted distributed protocols that can benefit from solutions and primitives that are initially designed for the nominal model. In Sections 4.4 to 4.6, we discuss in detail how to combine weighted voting and weight reduction to obtain extremely efficient weighted protocols without sacrificing resilience.

Ad-hoc solutions. There is a large body of work studying ad-hoc weighted cryptographic protocols [21, 23, 45, 51, 69, 84]. Compared to these works, the weight reduction approach studied in this chapter has a number of benefits, such as simplicity, efficiency, wider applicability, and a wider range of possible cryptographic assumptions. Moreover, in many cases, ad-hoc solutions can be combined with and benefit from weight reduction. In this chapter, we also study other, non-cryptographic, applications, such as erasure and error-corrected distributed storage and broadcast protocols.

Similar work by Benhamouda, Halevi, and Stambler. A recent work [23] mentioned a similar idea of reducing real weights to integers to construct *ramp* secret sharing. This project has been started and the first version of Swiper has been drafted before the online publication and without any knowledge of [23]. As the main focus of [23] is different, we believe that we do a much more in-depth exploration of this direction by studying different kinds of weight reduction problems and their applications beyond secret sharing, as well as providing much tighter bounds and implementing a solver that is not only linear in the worst case but also allocates very few tickets in empirical evaluations on real-world weight distributions.

Application in Aptos blockchain. Weight reduction has been recently used in the Aptos blockchain in their implementation of on-chain randomness [140].

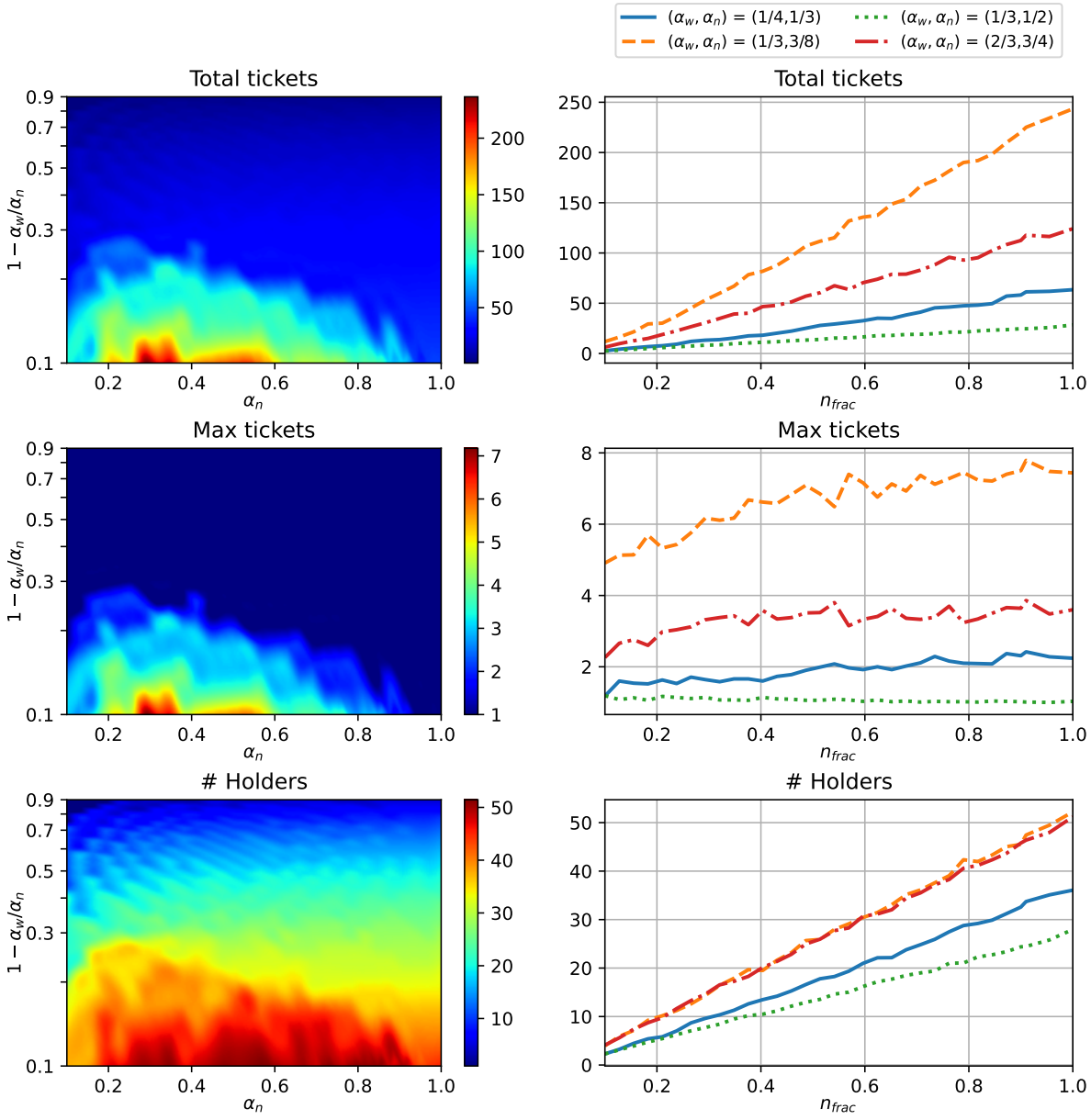

Figure 2: Experiment results using Aptos (100 samples per data point)

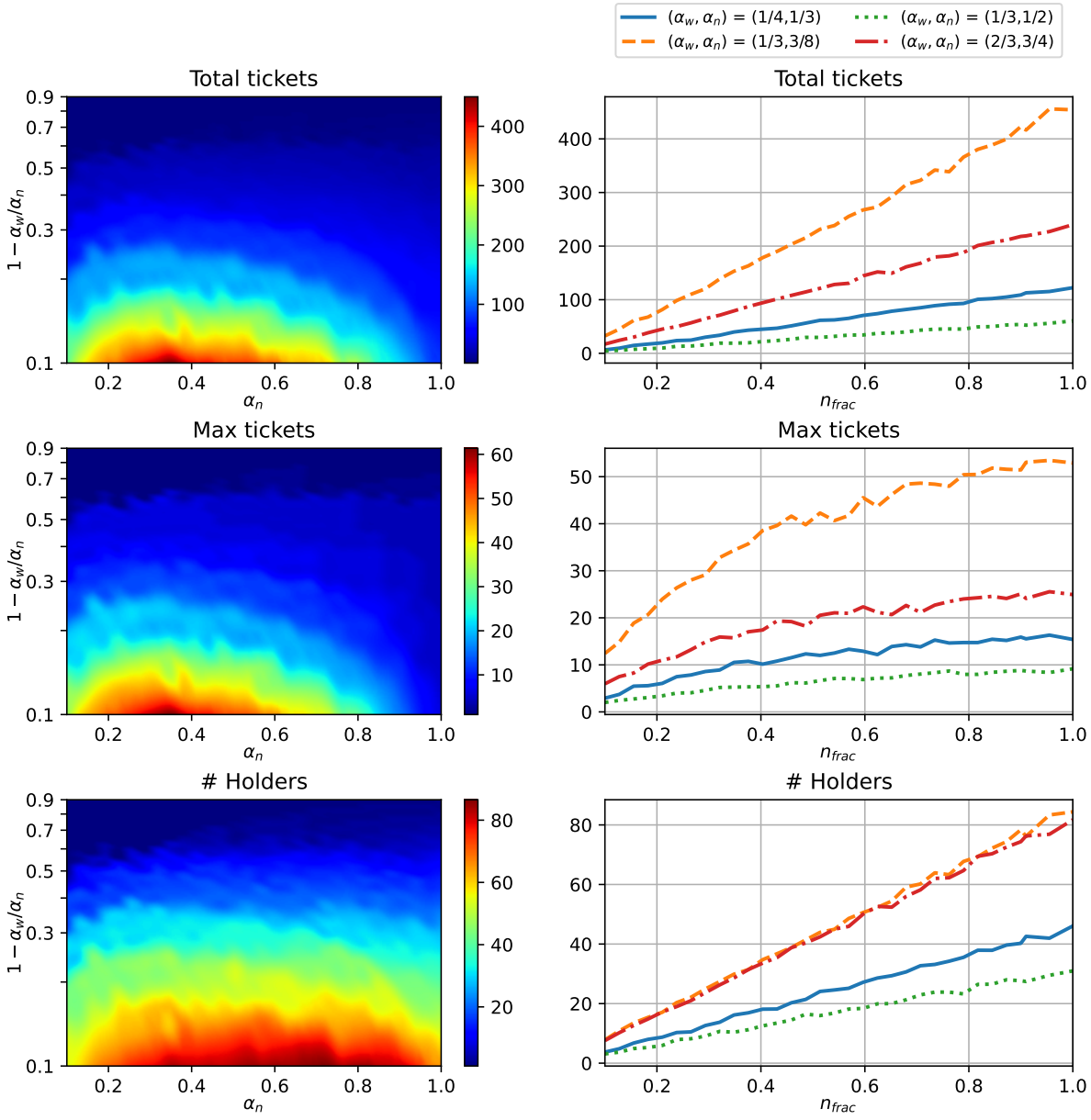

Figure 3: Experiment results using Tezos (100 samples per data point)

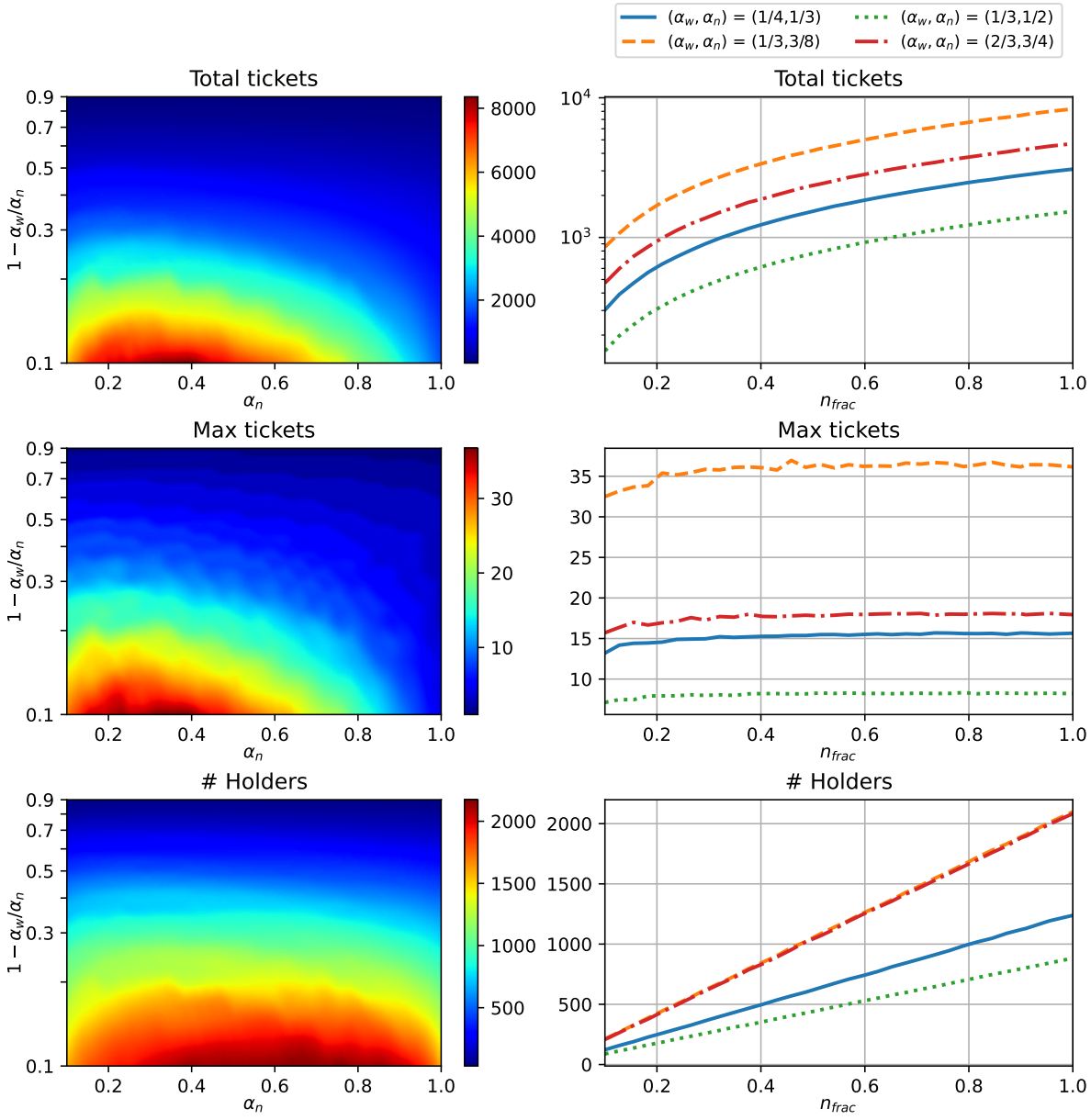

Figure 4: Experiment results using Filecoin (100 samples per data point)

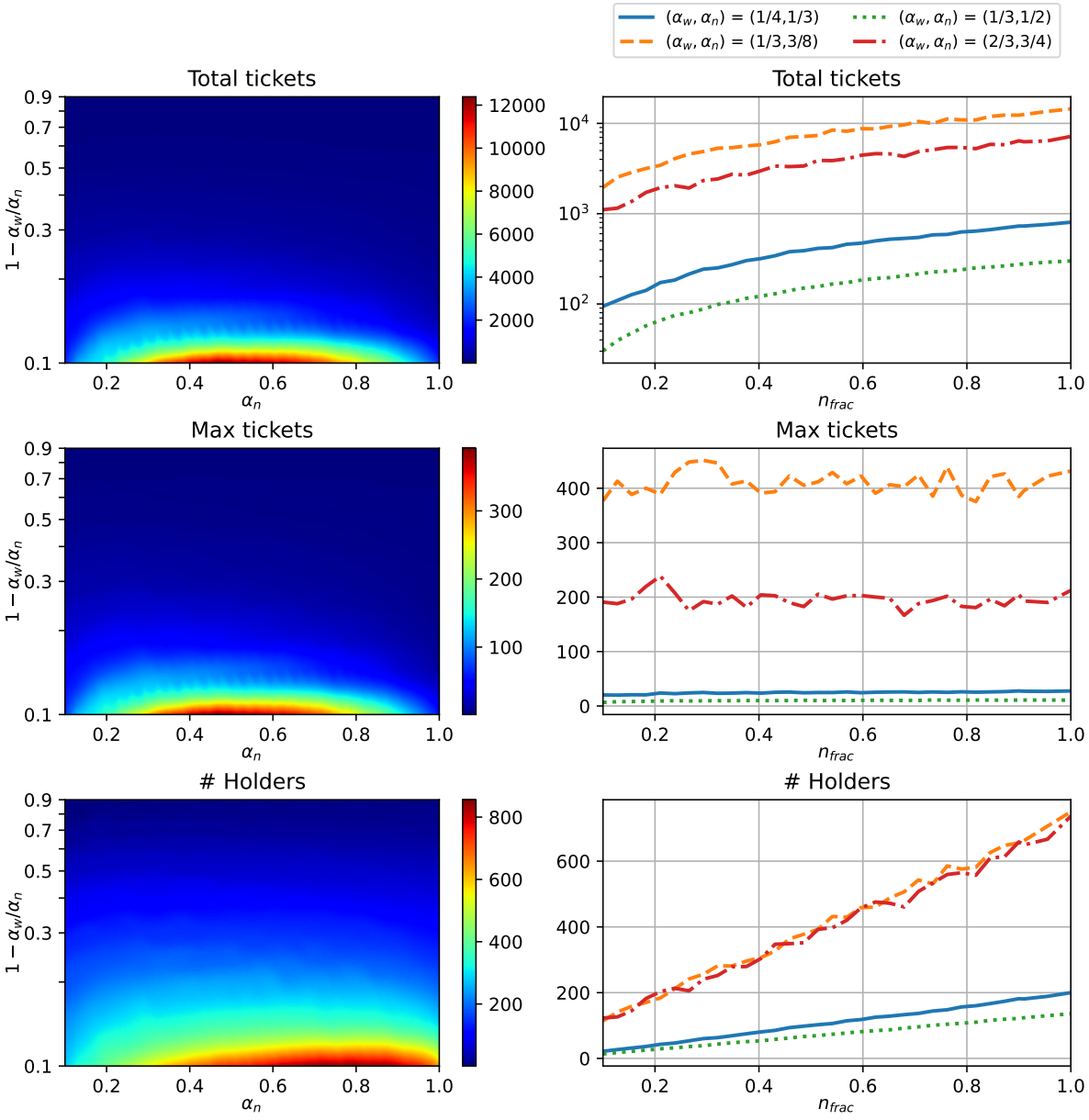

Figure 5: Experiment results using Algorand (100 samples per data point)

4.12 Concluding Remarks

In this chapter, we have presented a family of optimization problems called weight reduction that, to the best of our knowledge, has not been studied before. We provided practical protocols to find good, albeit not optimal, solutions to these problems. As we have shown, it allows us to obtain efficient implementations of many weighted distributed protocols.

We believe that weight reduction problems will play an important role in the future of blockchain systems as they become more sophisticated and the need for threshold cryptography as well as erasure coding and protocols like single secret leader election grows. At the time of writing, at least one major layer-1 blockchain has already integrated a version of Weight Separation for generating on-chain randomness.

We attempted the first systematic study of this family of problems, but there are still many important questions being left for future research.

Fairness. Weight reduction naturally leads to slight deviations in the relative weights of the participants. While in this chapter we focused on *safety* and *liveness* properties and showed that they can still be preserved, we did not consider any kind of *fairness* properties. However, we believe that, somewhat counterintuitively, some form of fairness can be preserved as well. To this end, we are considering two possible directions:

1. **Expected fairness:** In addition to deterministically assigned tickets, we can allocate some small number of tickets randomly so that each party gets exactly the same fraction of tickets as its fraction of weight *in expectation*. We believe that it can be done while still preserving safety and liveness *deterministically*, i.e., even in the worst case, when all the "random" tickets are received by the adversary.

2. **Integral fairness:** Similarly, one can imagine a *deterministic* protocol that provides fairness *over time*. In such a scheme, the ticket assignment will be updated periodically and each party will get exactly the right number of tickets *on average*, over a large enough period.

Incentives. One important aspect of proof-of-stake blockchains is the distribution of incentives, which should depend on the weight of each party. It is not immediately clear what is the right way to allocate incentives in a system where weight reduction is being applied.

Other applications. While we covered a wide range of applications, we believe that there must be others, including ones not related to distributed computing or applied cryptography.

Adversarial attacks. In this chapter, we study the "worst case" weight distributions by providing the upper bounds and the "organic case" by studying the real-world weight distributions. However, in practice, under an adversarial attack, the weight distribution will be a hybrid one: the weights of honest parties will be organic, but the weights of the adversarial parties may be redistributed maliciously. It is an interesting avenue for future work to study how much an adversary can affect the number of tickets (and, thus, the performance of the system) by redistributing their weight in a malicious manner.

Complexity and more precise bounds. Finally, there are still many theoretical questions about these problems. Do they have polynomial-time exact solutions? What are the lower bounds? Can we derive better upper bounds? Moreover, what are some other interesting and useful weight reduction problems, apart from the three defined in this chapter?

Chapter 5

Conclusion

In this dissertation, we discussed three major issues in the design of blockchain systems:

1. How to avoid unnecessary synchronization;
2. How to make consensus fast yet scalable;
3. How to make the protocols work with weighted participation.

At the end of each of the three technical chapters, in Sections 2.14, 3.9 and 4.12, we list open questions and potential directions for future research related to each of the three problems. Chapter 4 provokes the most open questions of the three as it establishes several optimization problems that, to the best of our knowledge, have not been systematically studied before, and demonstrates their applications to a large variety of problems in distributed computing and applied cryptography.

Furthermore, there are numerous other technological challenges that need to be addressed for blockchains to flourish and benefit the society in meaningful ways. The author of this dissertation continues to actively explore these challenges, working, among other things, on increasing throughput, scaling consensus protocols to more participants for greater decentralization, and enriching the fundamental capabilities of blockchains with cryptographic tools. Other interesting directions include the game-theoretic and socioeconomic side of blockchains, reliable and efficient storage, execution layer, sharding and other scaling solutions, multilayered system architecture (e.g., rollups and state channels), interoperability, and more.

Finally, in the development of technologies that create new capabilities, it is essential to take the responsible approach and invest effort in exploring ways to mitigate negative use-cases and promote the applications beneficial to the society as well as ensuring fair access to the technology and educating people to be able to use said technology safely. Among other things, with blockchains, we face the age-old questions of balancing privacy and personal freedoms with accountability and safety. However, this time, the problem is especially hard as public blockchains, by design, operate globally and, thus, require global cooperation to govern and oversee.

Chapter 6

Bibliography

[1] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil, 2018. arXiv:1803.05069v1 13 Mar 2018.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.

[3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, Italy, virtual, 2021. ACM.

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

[5] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *PODC*, 2021.

[6] Ittai Abraham, Ling Ren, and Zhuolun Xiang. Good-case and bad-case latency of unauthenticated byzantine broadcast: A complete categorization. *arXiv preprint arXiv:2109.12454*, 2021.

[7] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. Malicious security comes for free in consensus with leaders. *Cryptology ePrint Archive*, 2020.

[8] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. Brief announcement: Malicious security comes for free in consensus with leaders. In *PODC*, 2021.

[9] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

[10] Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires t+ 1 rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.

[11] Algoexplorer. Algorand stake distribution. `https://algoexplorer.io/top-accounts`, 2023. Accessed: 2023-03-28.

[12] Orestis Alpos, Christian Cachin, Giorgia Azzurra Marson, and Luca Zanolini. On the synchronization power of token smart contracts. In *ICDCS*, pages 640–651. IEEE, 2021.

[13] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.

[14] Aptos. White paper – the aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. Technical report, Aptos, 2022. URL: https://aptos.dev/aptos-white-paper/.

[15] Aptoscan. Aptos stake distribution. https://aptoscan.com/validators?ps=100&p=, 2023. Accessed: 2023-03-28.

[16] Thomas Attema, Ronald Cramer, and Matthieu Rambaud. Compressed sigma-protocols for bilinear circuits and applications to logarithmic-sized transparent threshold signature schemes. In *ASIACRYPT*, 2021.

[17] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[18] Sarah Azouvi and Marko Vukolić. Pikachu: Securing pos blockchains from long-range attacks by checkpointing into bitcoin pow using taproot. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, pages 53–65, Los Angeles, 2022. ACM.

[19] Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.

[20] Rida Bazzi and Maurice Herlihy. Clairvoyant state machine replication. *Information and Computation*, 285:104701, 2022.

[21] Amos Beimel and Enav Weinreb. Monotone circuits for monotone weighted threshold functions. *Information Processing Letters*, 97(1):12–18, 2006.

[22] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[23] Fabrice Benhamouda, Shai Halevi, and Lev Stambler. Weighted secret sharing from wiretap channels. Cryptology ePrint Archive, Paper 2022/1578, 2022. https://eprint.iacr.org/2022/1578. URL: https://eprint.iacr.org/2022/1578.

[24] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[25] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *ASIACRYPT*, 2018.

[26] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 12–24, New York, 2020. ACM.

[27] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, August 19–23, 2018, Proceedings, Part I 38*, pages 565–596, Santa Barbara, CA, USA, 2018. Springer.

[28] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, 2003.

[29] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[30] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of byzantine state-machine replication. *Distributed Computing*, pages 1–29, 2024.

[31] Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. In *DISC*, 2020.

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

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

[34] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[35] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, Washington, DC USA, 2002. ACM.

[36] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, August 19–23, 2001 Proceedings*, pages 524–541, Santa Barbara, California, USA, 2001. Springer.

[37] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132, Portland Oregon USA, 2000. ACM.

[38] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201, Orlando, Florida, USA, 2005. IEEE.

[39] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, San Diego California USA, 1993. ACM.

[40] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[41] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[42] Dario Catalano, Dario Fiore, and Emanuele Giunta. Adaptively secure single secret leader election from ddh. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 430–439, Salerno, Italy, 2022. ACM.

[43] Dario Catalano, Dario Fiore, and Emanuele Giunta. Efficient and universally composable single secret leader election from pairings. In *Public-Key Cryptography–PKC 2023: 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, May 7–10, 2023, Proceedings, Part I*, pages 471–499, Atlanta, GA, USA, 2023. Springer.

[44] Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-knowledge sets with short proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 433–450. Springer, 2008.

[45] Pyrros Chaidos and Aggelos Kiayias. Mithril: Stake-based threshold multisignatures, 2021.

[46] Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Strong byzantine agreement with adaptive word complexity. *arXiv preprint arXiv:2308.03524*, 2023.

[47] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Michael Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 277–290. ACM, 2009.

[48] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, pages 26–38. IEEE, 2020.

[49] Benoît Colson, Patrice Marcotte, and Gilles Savard. An overview of bilevel optimization. *Annals of operations research*, 153(1):235–256, 2007.

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

[51] Sourav Das, Philippe Camacho, Zhuolun Xiang, Javier Nieto, Benedikt Bunz, and Ling Ren. Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold, 2023.

[52] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, Virtual Event Republic of Korea, 2021. ACM.

[53] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2518–2534, San Francisco, CA, USA, 2022. IEEE.

[54] Yvo Desmedt. Threshold cryptosystems. In *Advances in Cryptology—AUSCRYPT'92: Workshop on the Theory and Application of Cryptographic Techniques Gold Coast, December 13–16, 1992 Proceedings 3*, pages 1–14, Queensland, Australia, 1993. Springer.

[55] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

[56] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

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

[58] Partha Dutta and Rachid Guerraoui. The inherent price of indulgence. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 88–97, 2002.

[59] Partha Dutta, Rachid Guerraoui, and Marko Vukolić. Best-case complexity of asynchronous byzantine consensus. Technical report, EPFL, 2005.

[60] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

[61] Jose M. Falerio, Sriram K. Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 125–134. ACM, 2012.

[62] Filfox. Filecoin stake distribution. `https://filfox.info/en/ranks/power`, 2023. Accessed: 2023-03-28.

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

[64] Fish. Tezos stake distribution. `https://tezos.fish/leaderboard/all`, 2023. Accessed: 2023-03-28.

[65] Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Brief announcement: Asynchronous randomness and consensus without trusted setup. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 103–105, 2022.

[66] Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Distributed randomness from approximate agreement. In *36th International Symposium on Distributed Computing*, 2022.

[67] Luciano Freitas, Andrei Tonkikh, Adda-Akram Bendoukha, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, and Petr Kuznetsov. Homomorphic sortition – single secret leader election for PoS blockchains. Cryptology ePrint Archive, Paper 2023/113, 2023. URL: `https://eprint.iacr.org/2023/113`.

[68] Luciano Freitas de Souza, Andrei Tonkikh, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, Nicolas Quero, and Petr Kuznetsov. Randsolomon: Optimally resilient random number generator with deterministic termination. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[69] Sanjam Garg, Abhishek Jain, Pratyay Mukherjee, Rohit Sinha, Mingyuan Wang, and Yinuo Zhang. Cryptography with weights: Mpc, encryption and signatures, 2022.

[70] Giuliano Garrammone. On decoding complexity of reed-solomon codes on the packet erasure channel. *IEEE Communications Letters*, 17(4):773–776, 2013.

[71] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *FC*, 2022.

[72] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, Pacific Grove California USA, 1979. ACM.

[73] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. No-commit proofs: Defeating livelock in bft. *Cryptology ePrint Archive*, 2021.

[74] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *DSN*, 2019.

[75] L.M Goodman. White paper – tezos: a self-amending crypto-ledger. Technical report, Tezos, 2014. URL: `https://tezos.com/whitepaper.pdf`.

[76] Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[77] Rachid Guerraoui and Petr Kuznetsov. *Algorithms for Concurrent Systems*. EPFL press, 2018.

[78] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In Peter Robinson and Faith Ellen, editors, *PODC*, pages 307–316, 2019.

[79] Saurabh Gupta. A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing. Master's thesis, Arizona State University, USA, 2016.

[80] James Hendricks, Gregory R Ganger, and Michael K Reiter. Verifying distributed erasure-coded data. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 139–146, Portland Oregon USA, 2007. ACM.

[81] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[82] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. `doi:10.1145/78969.78972`.

[83] Damien Imbs and Michel Raynal. Trading off t-resilience for efficiency in asynchronous byzantine reliable broadcast. *Parallel Processing Letters*, 26(04):1650017, 2016.

[84] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[85] Aayush Jain, Peter MR Rasmussen, and Amit Sahai. Threshold fully homomorphic encryption, 2017.

[86] Mohammad M. Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and resilient hotstuff protocol, 2021. `arXiv:2010.11454`.

[87] Jørn Justesen. On the complexity of decoding reed-solomon codes (corresp.). *IEEE transactions on information theory*, 22(2):237–238, 1976.

[88] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3465084.3467905`.

[89] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[90] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.

[91] Klaus Kursawe. Optimistic byzantine agreement. In *21st Symposium on Reliable Distributed Systems (SRDS 2002), 13-16 October 2002, Osaka, Japan*, pages 262–267. IEEE Computer Society, 2002.

[92] Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer. *Distributed Computing*, 36(3):349–371, 2023.

[93] Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. *Distributed Computing*, 35(6):477–502, 2022.

[94] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast byzantine consensus. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 343–353, 2021.

[95] Protocol Labs. White paper – filecoin: A decentralized storage network. Technical report, Protocol Labs, 2017. URL: `https://filecoin.io/filecoin.pdf`.

[96] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

[97] Leslie Lamport. The Part-Time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[98] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[99] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[100] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 2010.

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

[102] Leslie B Lamport. Generalized paxos, April13 2010. US Patent 7,698,465. URL: `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-33.pdf`.

[103] Christoph Lenzen and Julian Loss. Optimal clock synchronization with signatures. In *PODC*, 2022.

[104] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012. URL: `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li`.

[105] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, Philadelphia, PA, USA, 1977.

[106] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[107] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[108] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine paxos. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 402–411, 2004.

[109] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. In *DSN*, 2005.

[110] Silvio Micali. ALGORAND: the efficient and democratic ledger, 2016. URL: `http://arxiv.org/abs/1607.01341`, `arXiv:1607.01341`.

[111] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, Vienna, Austria, 2016. ACM.

[112] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[113] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[114] Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine SMR. In *DISC*, 2020.

[115] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved Extension Protocols for Byzantine Broadcast and Agreement. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2020.28`, `doi:10.4230/LIPIcs.DISC.2020.28`.

[116] Kamilla Nazirkhanova, Joachim Neu, and David Tse. Information dispersal with provable retrievability for rollups, 2021.

[117] Kazuo Ohta and Tatsuaki Okamoto. Multi-signature schemes secure against active insider attacks. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 82(1):21–31, 1999.

[118] Miguel Pires, Srivatsan Ravi, and Rodrigo Rodrigues. Generalized paxos made byzantine (and less complex). *Algorithms*, 11(9):141, 2018.

[119] Michael O Rabin. Randomized byzantine generals. In *24th annual symposium on foundations of computer science (sfcs 1983)*, pages 403–409, Tucson, Arizona, USA, 1983. IEEE.

[120] Michael O Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989.

[121] Mayank Raikwar and Danilo Gligoroski. Sok: Decentralized randomness beacon protocols. In *Information Security and Privacy: 27th Australasian Conference, ACISP 2022, November 28–30, 2022, Proceedings*, pages 420–446, Wollongong, NSW, Australia, 2022. Springer.

[122] Matthieu Rambaud, Andrei Tonkikh, and Mark Abspoel. Linear view change in optimistically fast bft. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, pages 67–78, 2022.

[123] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[124] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques, May 14–18, 2000 Proceedings 19*, pages 207–220, Bruges, Belgium, 2000. Springer.

[125] Jakub Sliwinski, Yann Vonlanthen, and Roger Wattenhofer. Consensus on demand. In *Stabilization, Safety, and Security of Distributed Systems: 24th International Symposium, SSS 2022, Clermont-Ferrand, France, November 15–17, 2022, Proceedings*, pages 299–313. Springer, 2022.

[126] Jakub Sliwinski and Roger Wattenhofer. ABC: asynchronous blockchain without consensus. *CoRR*, abs/1909.10926, 2019. URL: `http://arxiv.org/abs/1909.10926`, `arXiv:1909.10926`.

[127] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *DISC*, 2008.

[128] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2705–2718, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3548606.3559361`.

[129] Douglas R. Stinson and Reto Strobl. Provably secure distributed schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy*, ACISP '01, page 417–434, Berlin, Heidelberg, 2001. Springer-Verlag.

[130] Xiao Sui, Sisi Duan, and Haibin Zhang. Marlin: Two-phase bft with linearity. In *DSN*, 2022.

[131] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan-Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 877–893. IEEE, 2020.

[132] Andrei Tonkikh and Luciano Freitas. Swiper: a new paradigm for efficient weighted distributed protocols. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 283–294, 2024.

[133] Andrei Tonkikh, Pavel Ponomarev, Petr Kuznetsov, and Yvonne-Anne Pignolet. Cryptoconcurrency:(almost) consensusless asset transfer with shared accounts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1556–1570, 2023.

[134] Wikipedia. List of blockchains — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/List_of_blockchains`, 2024. [Online; accessed 18-October-2024].

[135] Wikipedia. Proof of stake — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Proof_of_stake`, 2024. [Online; accessed 18-October-2024].

[136] Wikipedia. Proof of work — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Proof_of_work`, 2024. [Online; accessed 18-October-2024].

[137] Wikipedia. Smart contract — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Smart_contract`, 2024. [Online; accessed 18-October-2024].

[138] Wikipedia. Unspent transaction output — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Unspent_transaction_output`, 2024. [Online; accessed 18-October-2024].

[139] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[140] Zhuolun Xiang, Sourav Das, Zekun Li, Zhoujun Ma, and Alexander Spiegelman. The latency price of threshold cryptosystem in blockchains. *arXiv preprint arXiv:2407.12172*, 2024.

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

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

ECOLE
DOCTORALE

Titre : Le calcul distribué pour la blockchain et plus encore

Mots clés : Systèmes distribués, Tolérance aux pannes byzantines, Blockchain, Consensus, Réplication de la machine d'état, Systèmes asynchrones

Résumé : Dans cette thèse, nous abordons trois défis majeurs dans la conception des systèmes de blockchain en particulier et des systèmes distribués tolérants aux pannes à grande échelle en général. Ce travail vise à améliorer directement la performance de tels systèmes, ainsi qu'à fournir des outils utiles pour le développement futur d'algorithmes distribués.
Premièrement, nous explorons les limites de ce qui peut être réalisé avec une synchronisation minimale en concevant CryptoConcurrency—un système de *transfert d'actifs* qui, au lieu d'ordonner totalement toutes les requêtes des utilisateurs, traite les requêtes concurrentes en parallèle *autant que possible*. Contrairement à d'autres systèmes similaires, dans CryptoConcurrency, nous permettons aux utilisateurs d'avoir des comptes partagés et ne faisons pas l'hypothèse irréaliste qu'un compte d'utilisateur honnête n'est jamais accédé simultanément depuis deux dispositifs. CryptoConcurrency explore de nouveaux terrains théoriques en abordant les conflits de transactions de manière dynamique et non par paires, permettant aux propriétaires de chaque compte de choisir indépendamment leur mécanisme préféré de résolution de conflits.
Ensuite, nous améliorons la performance du *consensus*—le problème de synchronisation au cœur de la plupart des systèmes distribués pratiques. Nous construisons le premier protocole de consensus qui parvient à combiner deux propriétés souhaitables : une terminaison extrêmement rapide dans des conditions favorables et une récupération gracieuse lorsque ces conditions ne sont pas remplies. La conception implique un nouveau type de preuves cryptographiques, avec une implémentation pratique et efficace.
Enfin, nous nous attaquons au problème de la conception de protocoles distribués efficaces avec une *participation pondérée*. À cette fin, nous définissons plusieurs nouveaux problèmes d'optimisation, liés à la réduction ou, en d'autres termes, à la quantification des poids des participants d'une manière qui préserve d'importantes propriétés structurelles. Nous montrons comment les appliquer pour créer des variantes pondérées d'un large éventail de protocoles distribués avec très peu de surcharge par rapport à leurs homologues dans le modèle non pondéré plus simple. Pour ces problèmes d'optimisation, nous prouvons des bornes supérieures, fournissons un solveur pratique open-source approximatif qui satisfait ces bornes, et effectuons une étude empirique sur les distributions de poids provenant de systèmes de blockchain réels.

Title : Distributed computing for blockchains and beyond

Keywords : Distributed systems, Byzantine fault tolerance, Blockchain, Consensus, State Machine Replication, Asynchronous Systems

Abstract : In this dissertation, we address three major challenges in the design of blockchain systems in particular and large-scale fault-tolerant distributed systems in general. This work aims at improving the performance of such systems directly, as well as providing useful tools for future development of distributed algorithms.
First, we explore the limits of what can be done with minimal synchronization by designing CryptoConcurrency—an *asset transfer* system that, instead of totally ordering all users' requests, processes concurrent requests in parallel *as much as possible*. Unlike other similar systems, in CryptoConcurrency, we allow the users to have shared accounts and do not make the unrealistic assumption that an honest user's account is never accessed from two devices concurrently. CryptoConcurrency explores novel theoretical grounds by addressing transaction conflicts in a dynamic and non-pairwise manner, allowing the owners of each account to independently choose their preferred mechanism for conflict resolution.
Then, we improve the performance of *consensus*—the synchronization problem at the heart of most practical distributed systems. We build the first consensus protocol that manages to combine two desirable properties: extremely fast termination in favorable conditions and graceful recovery when such conditions are not met. The design involves a novel type of cryptographic proofs, with an efficient practical implementation.
Finally, we set out to tackle the problem of designing efficient distributed protocols with *weighted participation*. To this end, we define several new optimization problems, related to reducing or, in other words, quantizing the weights of the participants in a way that preserves important structural properties. We show how to apply them to make weighted-model variants of a large class of distributed protocols with very little overhead compared to their counterparts in the simpler non-weighted model. For these optimization problems, we prove upper bounds, provide a practical open-source approximate solver that satisfies these upper bounds, and perform an empirical study on the weight distributions from real-world blockchain systems.

Institut Polytechnique de Paris
91120 Palaiseau, France