UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Probabilistic Chained Blockchain Consensus

André David dos Santos

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor Alysson Neves Bessani
Prof. Doutor Hasan Heydari

2025

Acknowledgments

First and foremost, I would like to express my sincere appreciation to all those who have supported me in the completion of this thesis.

I would like to begin by expressing my deepest gratitude to both my supervisor, Prof. Alysson Bessani, and co-advisor Prof. Hasan Heydari for their invaluable guidance, continuous support, and availability throughout the course of this research. Their insightful feedback greatly enriched both this thesis and my academic journey.

I would also like to acknowledge the financial support provided by the Fundação para a Ciência e a Tecnologia (FCT). This work was supported by FCT through the project SMaRtChain, ref. 2022.08431.PTDC (https://doi.org/10.54499/2022.08431.PTDC), and the LASIGE Research Unit, ref. UID/00408/2025 (https://doi.org/10.54499/UID/00408/2025).

Most importantly, I want to show my gratitude to my family for their continuous support, love and understanding during stressful moments.

Lastly, I would like to thanks Prof. Vinicius Cogo for its indispensable help during the experimental phase.

Thanks to everyone involved in this work. Without everyone's support, this thesis would not have been fully completed.

Resumo

Sistemas distribuídos são constituídos por uma rede de múltiplas máquinas independentes (réplicas) que interagem entre si e constituem a base de inúmeros serviços modernos altamente complexos, desde infraestruturas críticas até aplicações financeiras descentralizadas. A grande complexidade inerente a estes sistemas reside na necessidade de assegurar que, apesar de falhas e incertezas na comunicação, todas as réplicas corretas cooperem de forma coordenada e robusta para fornecer um serviço fiável e consistente. De modo a garantir consistência, é utilizada uma primitiva conhecida como consenso, que define um problema no qual, um conjunto de réplicas independentes deve concordar de maneira segura num único valor proposto, mesmo quando algumas destas réplicas são comprometidas ou agem de forma arbitrária (falhas bizantinas). Sistemas tolerantes a falhas bizantinas (*Byzantine Fault Tolerance* – BFT), são resilientes a este tipo de comportamentos adversariais e garantem consenso entre as múltiplas réplicas corretas do sistema.

Uma das técnicas mais utilizadas para implementar sistemas tolerantes a faltas consiste na replicação de máquinas de estado (*State Machine Replication* – SMR), que define um serviço replicado no qual as múltiplas réplicas mantêm cópias idênticas e sincronizadas de uma máquina de estados determinística. Para assegurar que todas as réplicas evoluem de forma coordenada, é necessário garantir que as operações que manipulam o seu estado sejam executadas pela mesma ordem. Neste sentido, implementar SMR é equivalente a resolver o problema de difusão com ordem total (*Total order broadcast*), no qual réplicas devem entregar o mesmo conjunto de mensagens pela mesma ordem, mesmo na presença de falhas. Em ambos os casos, para garantir a consistência e fiabilidade do serviço fornecido, estes sistemas recorrem a protocolos de consenso, que especificam o comportamento que as réplicas do sistema devem assumir de forma a chegarem a um acordo na presença de faltas bizantinas. A implementação destes protocolos tolerantes a faltas bizantinas exige que o sistema disponha de pelo menos $3f + 1$ réplicas em ambientes parcialmente síncronos, de forma a tolerar um máximo de f faltas. Os protocolos de *blockchain*, como Bitcoin e Ethereum, representam uma evolução dos protocolos BFT tradicionais. O termo *blockchain* é utilizado para descrever uma sequência dos valores acordados em blocos encadeados, que apenas permitem anexação. Cada bloco inclui um conjunto de transações e uma referência criptográfica ao bloco anterior. A estrutura de dados subjacente a uma *blockchain* é geralmente designada por *ledger* e representa um registo persistente e imutável de todas as transações confirmadas.

Os protocolos de consenso tolerantes a faltas bizantinas constituem um dos principais componentes para a construção de sistemas distribuídos e, em particular, para a tecnologia *blockchain*. Estes protocolos são tradicionalmente concebidos a partir de pressupostos extremamente pessimistas, de forma a garantir propriedades de segurança e vivacidade mesmo em cenários nos quais participantes maliciosos, também conhecidos como réplicas bizantinas, podem adotar comportamentos arbitrários e potencialmente prejudiciais à resolução do problema de consenso. Embora esta abordagem garanta robustez, ela impõe restrições significativas em termos de desempenho e escalabilidade, sobretudo em sistemas compostos por um grande número de réplicas.

De maneira geral, os principais protocolos BFT atuais enfrentam um dilema fundamental: de um lado, existem protocolos que optam por baixa latência, mas com o custo de apresentarem complexidade quadrática na troca de mensagens (todas as réplicas enviam mensagens para todas as réplicas); de outro, encontram-se os protocolos que se focam em reduzir a complexidade de mensagens, mas adicionam fases de comunicação extra, o que aumenta significativamente o tempo de resposta fim a fim (latência). O protocolo Practical Byzantine Fault Tolerance (PBFT), por exemplo, representa a primeira categoria de protocolos, que atingem o número ideal de fases de comunicação (três), mas sacrificam eficiência ao recorrer a um padrão de troca de mensagens com complexidade quadrática ou superior. O protocolo HotStuff, por outro lado, adota o caminho oposto, preferindo uma estratégia que obtém uma complexidade de mensagens linear, mas que ao mesmo tempo aumenta o número de fases de comunicação e consequentemente a latência de finalização de blocos.

Com o objetivo de solucionar este problema, surgiu recentemente um novo protocolo denominado ProBFT, um protocolo probabilístico de consenso bizantino desenvolvido para sistemas parcialmente síncronos e permissionados. ProBFT opta por relaxar os pressupostos pessimistas tradicionais e, em contrapartida, alcança uma complexidade subquadrática de mensagens e uma latência ideal num cenário sem adversidades, de apenas três etapas de comunicação. Ao relaxar os pressupostos tradicionais, o ProBFT deixa de assegurar propriedades de segurança e vivacidade de forma determinística, passando a garanti-las apenas com alta probabilidade. No entanto, apesar das suas contribuições teóricas, o ProBFT apresenta dois problemas relevantes: (1) o protocolo nunca foi implementado nem avaliado experimentalmente e (2) a sua especificação resolve apenas o problema de consenso de instância única (*single-shot consensus*), o que o torna inadequado para sistemas que exigem a manutenção e replicação de cadeias de blocos interligados contendo transações, como nas *blockchain* modernas.

Com o objetivo de superar essas limitações, nesta tese, integramos os mecanismos fundamentais do ProBFT a um novo protocolo, Practical Simplex, um protocolo de consenso BFT que constitui uma adaptação do protocolo Simplex, que destaca-se pela a sua estratégia simples de ordenação de blocos de transações. A versão original do protocolo Simplex apresenta uma limitação crítica que inviabiliza a sua aplicação em cenários práticos: em cada iteração, cada réplica deve enviar uma mensagem que contém a totalidade da sua *blockchain* a todas as outras réplicas do sistema. Este requisito conduz a um crescimento indefinido da complexidade de comunicação do protocolo, sobrecarregando a rede com mensagens progressivamente maiores à medida que o protocolo avança. O Practical Simplex foi desenvolvido para superar esta limitação, eliminando a necessidade de comunicação crescente e tornando o protocolo mais eficiente e viável em sistemas distribuídos reais. Além disso, este novo protocolo apresenta-se como uma ótima fundação para a integração dos mecanismos probabilísticos presentes no ProBFT, o que resulta no desenvolvimento do ProSimplex, um protocolo de consenso probabilístico, baseado nos principais mecanismos do ProBFT, que implementa uma máquina de replicação de estado e apresenta uma complexidade de mensagens subquadrática juntamente com uma latência ótima de três passos de comunicação nos casos ideais. O novo protocolo, ProSimplex, elimina também a necessidade do ProBFT de um subprotocolo de *view-change* para o avanço de iterações do protocolo.

Os protocolos ProSimplex e Practical Simplex foram implementados em Rust e avaliados experimentalmente. Ambos foram comparados com Jolteon, uma evolução do protocolo HotStuff, que apresenta também uma complexidade linear de mensagens e reduz a latência e o número de fases de comunicação do protocol original. Os resultados experimentais mostram que o ProSimplex apresenta melhor escalabilidade do que o Practical Simplex e Jolteon em termos de desempenho (transações por segundo) para sistemas com um maior número de réplicas, apesar de exibir uma latência de finalização ligeiramente superior ao Practical Simplex devido à sua natureza probabilística. Para sistemas de menor dimensão, o

custo adicional das operações criptográficas do ProSimplex reduz o desempenho relativamente aos outros protocolos. Por outro lado, o protocolo Jolteon apresenta a maior latência de finalização devido ao maior número de fases de comunicação. Adicionalmente, os resultados demonstram que o desempenho dos três protocolos é mais próximo do que seria esperado, consequência da sobrecarga comum associada à disseminação de grandes mensagens compostas por blocos que contêm muitas transações de dimensão considerável.

Palavras-chave: Tolerância a Faltas Bizantinas, Blockchain, Consenso Probabilístico, Escalabilidade, Sistemas Distribuídos

Abstract

Byzantine Fault Tolerant (BFT) consensus protocols are a fundamental building block for blockchain and distributed systems, traditionally designed under pessimistic assumptions that guarantee safety and liveness even under arbitrary adversarial behavior. While robust, this approach often limits performance and scalability, particularly in large replica systems. Existing protocols face a trade-off: achieving low latency with quadratic message complexity (e.g. PBFT, Simplex) or, reducing message complexity at the expense of additional communication steps and increased latency (e.g. HotStuff, Jolteon).

Building on this foundation, ProBFT, a recent probabilistic BFT consensus protocol for partially synchronous systems, achieves sub-quadratic message complexity and optimal good-case latency of three communication steps by relaxing pessimistic assumptions while providing strong theoretical guarantees. However, ProBFT has never been implemented and only solves the single-shot consensus problem. To overcome this drawback, we develop Practical Simplex, an adaptation of Simplex that solves its main practical limitation: the unbounded growth of communication complexity required to guarantee liveness. Then, ProBFT's core mechanisms are integrated into Practical Simplex. This integration results in ProSimplex, a probabilistic chained consensus protocol that preserves Simplex's simple approach to consensus while leveraging probabilistic quorums to achieve sub-quadratic message and communication complexity, improving scalability.

ProSimplex and Practical Simplex are implemented and experimentally evaluated, and compared with Jolteon, a state-of-the-art chained consensus protocol. Experimental results show that ProSimplex scales better than Practical Simplex and Jolteon in terms of throughput for larger systems while exhibiting higher finalization latency when compared with Practical Simplex due to its probabilistic nature. For smaller systems, the cryptographic overhead of ProSimplex reduces throughput relative to the other protocols. Jolteon exhibits the highest finalization latency due to having more communication steps. Additionally, the results show that the throughput of all three protocols is closer than expected, due to the shared overhead of broadcasting large block proposals.

Keywords: Byzantine Fault Tolerance, Blockchain, Probabilistic Consensus, Scalability, Distributed Systems

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In distributed systems, a collection of machines appearing to behave like a single coherent system, work together to ensure state synchronization, redundancy and resilience with the main goal of hiding complexity and failures from end users of an application. We call Byzantine fault tolerance (BFT) [24], the resilience of a distributed system to failures of its participants, where faulty participants can behave arbitrarily and potentially deviate from the system's specifications. Blockchain protocols build upon these foundations, applying BFT principles in decentralized environments to implement a shared ledger that records transactions in blocks [27]. To add a new block to this ledger, participants must first agree through a consensus process. Consensus is a fundamental problem and building block for structuring reliable and fault-tolerant distributed systems where participants must reach an agreement on a single common value (typically execution order of requests or transactions) even in the presence of faults. Consensus protocols are used to replicate the system's state across multiple participants to ensure correctness and redundancy.

The design and implementation of BFT consensus protocols in practical networks presents itself as challenging. The first practical BFT consensus protocols were originally conceived, a typical target system size was $n = 4$ or $n = 7$ (n being the number of participants on a system), deployed on a local-area network [6]. However, the renewed interest in Byzantine fault-tolerance brought about by its application to blockchains now demands solutions that can scale to a much larger n able to be deployed on wide-area networks [35]. BFT consensus protocols that operate in partial synchrony, typically, require a minimum of $3f + 1$ participants (where f is the maximum number of faults tolerated by the system) which can be expensive in terms of hardware. Most classical BFT consensus protocols require all-to-all communication in some phases and increasing n can lead to network saturation in large-scale systems and bigger delays when awaiting for responses from a quorum of participants in the case of protocols that satisfy responsiveness. In BFT consensus protocols, responsiveness refers to the system's ability to make progress without relying on fixed timeouts, adapting instead to actual network conditions. Responsive protocols (e.g., [2, 9, 30, 35, 15, 12]) are designed to make progress as fast as the network allows, avoiding unnecessary delays and reducing latency. From a practical standpoint, BFT consensus protocols are often very complex, both in terms of understanding and implementation, particularly when accounting for all the corner cases that can arise when dealing with partially synchronous network models. Certain efforts (e.g., Streamlet [10], Simplex [9]) have specifically aimed to improve the comprehensibility and clarity of protocols' designs. Nevertheless, this complexity may sometimes lead to cases where certain hidden aspects of a protocol may ultimately violate safety or liveness guarantees.

1.1 Motivation

Traditionally, BFT consensus protocols (e.g., PBFT [8], HotStuff [35]) assume an extremely pessimistic approach when dealing with Byzantine adversaries, ensuring safety and liveness properties even when Byzantine participants behave completely arbitrarily under worst-case scenarios. The goal of protocols designed under this pessimistic approach is to make immutable decisions based on the opinions of a quorum of participants that is sufficiently large to ensure that any two decision-making quorums intersect in at least one correct participant.

Ensuring quorum overlaps poses inherent challenges in achieving both resource efficiency and high performance [3]. Some protocols (e.g., PBFT) approach this conflict by opting for low latency and applying message-exchange patterns with quadratic message complexity. However, this can be prohibitively expensive, especially for BFT systems with a large number of replicas. Other protocols (e.g., HotStuff) aim at reducing message complexity at the cost of adding extra communication steps. Unfortunately, this approach leads to increased end-to-end response times [3]. Despite this, in practical scenarios, adversaries are not as capable as the ones assumed in these protocols and it may suffice to ensure safety and liveness properties with high probability. Recently, a BFT leader-based probabilistic consensus protocol was developed for permissioned partially synchronous systems, Probabilistic Byzantine Fault Tolerance (ProBFT [3]), that guarantees these properties with high probability while requiring sub-quadratic message complexity and the optimal good-case latency of three communication steps [1], by relaxing the typical pessimistic assumptions.

1.2 Problem Statement

The ProBFT protocol has never been implemented or evaluated experimentally and has the drawback of only solving the single-shot consensus problem.

The objective of this thesis is to overcome ProBFT's limitation by integrating its core mechanisms to a more recent Byzantine chained consensus protocol, Simplex, that introduces a simpler approach for ordering blocks of transactions compared to PBFT or HotStuff due to not relying on a view-change sub-protocol. Specifically, by leveraging ProBFT's main ideas, this project aims to construct a responsive BFT state machine replication (SMR) and streamlined blockchain consensus protocol inspired by chained consensus protocols, that eliminates ProBFT's necessity of a view-change sub-protocol and is able to replicate a chain of blocks across multiple participants. Additionally, this new protocol should be evaluated and compared experimentally with Jolteon [15], a state-of-the-art chained consensus protocol with linear message complexity that improves the original HotStuff protocol by reducing the number of necessary communication steps for block finalization in the optimal common case from 7 to 5 and reduces HotStuff's finalization latency.

1.3 Contributions

In this thesis, we present and implement ProSimplex, a probabilistic chained consensus protocol based on the core mechanisms of single-shot ProBFT that achieves sub-quadratic message complexity and responsiveness. We also present an experimental evaluation and overview of our protocol and compare its performance and finalization latency with Jolteon. We show that such a protocol is able to achieve better latency and throughput than Jolteon, even with sub-quadratic message complexity, for a sufficiently large number of participants in the system. Furthermore, we additionally present and implement a practical

adaptation of Simplex used as the foundation for ProSimplex. This adaptation solves the practical issues of Simplex. These issues will be addressed in subsequent sections of this report.

In summary, the contributions of this thesis are the following:

- We identify the main limitations of Simplex.
- We develop a practical adaptation of Simplex that addresses these limitations, called Practical Simplex.
- We present ProSimplex, a probabilistic chained consensus protocol that integrates ProBFT's core mechanisms into Practical Simplex.
- We implement three protocols, Practical Simplex, ProSimplex and Jolteon, and experimentally evaluate the performance and finalization latency of each one, while simultaneously comparing the obtained results.

1.4 Document Organization

The document is organized as follows:

- **Chapter 2** provides the necessary background on SMR, BFT consensus, probabilistic consensus, blockchain and introduces important aspects of consensus protocols.
- **Chapter 3** presents the related work, covering state-of-the-art BFT consensus protocols with a particular focus on ProBFT.
- **Chapter 4** addresses the practical limitations of the Simplex protocol and introduces Practical Simplex, an adaptation that overcomes these limitations.
- **Chapter 5** introduces ProSimplex, a BFT probabilistic chained consensus protocol based on the core mechanisms of ProBFT.
- **Chapter 6** details the implementation of the Practical Simplex and ProSimplex protocols.
- **Chapter 7** evaluates the performance of Practical Simplex and ProSimplex and compares them with Jolteon.
- **Chapter 8** concludes the developed work and outlines potential improvements for future work.

Chapter 2

Background

This chapter provides the necessary background on distributed system fundamentals and the core ideas underlying BFT consensus protocols by tackling and introducing some concepts which are essential for understanding the context and challenges addressed in the subsequent chapters. Specifically, it introduces related concepts such as state machine replication, consensus and probabilistic consensus, blockchain, communication and fault models, responsiveness and at last, what we consider as block and finalization times.

Throughout the 21st century, much development has been made since the first well-known fault tolerant protocols, Paxos [23] and PBFT [8]. In distributed systems, faults are an unexpected or abnormal behavior in a component that can lead to an error or failure. Faults can manifest in different ways impacting the performance, reliability or correctness of the system. Resilient distributed systems continue operating well even in the presence of faults.

The two main types of faults that define fault tolerant systems are crash and Byzantine faults ([20], [25]). Crash faults refer to processes in a system that stop making progress and no longer execute their intended function. Examples of protocols that assume this fault model are Paxos , Zookeeper [19] and Raft [28]. On the other hand, faulty processes that deviate from the prescribed protocol in any form are called Byzantine faults. This type of faults was first introduced by Lamport in the Byzantine Generals Problem [24]. Faulty processes may be controlled by an adversary while non-faulty processes, also called correct processes, are faithful to the description of a protocol.

In distributed systems, the system model defines the assumptions about the underlying environment in which consensus must be achieved. The system model include aspects such as message delivery and computation timing delays, communication reliability, and adversarial behavior. Consensus protocols are designed under one of the following communication models, synchronous, asynchronous or partially synchronous.

In synchronous distributed systems, there is a known upper bound on the time it takes for messages to be delivered. This assumption simplifies the design and correctness analysis of BFT protocols. Despite this, to achieve a high performance protocol, this upper bound should be small, however, is is not realistic to assume that communication delays are always less than a small bound. Asynchronous systems lack any timing guarantees, making it impossible to distinguish between slow and faulty processes. This communication model is not appropriate for designing deterministic consensus protocols because it makes it impossible to obtain safety and liveness in a deterministic way according to the FLP impossibility theorem [14]. This is the reason why the majority of Byzantine fault tolerant consensus protocols assume partial synchrony [13]. In a partially synchronous model [9], [3], the system is initially asynchronous but eventually becomes synchronous. Consequently, timing bounds are unknown until an unknown global

stabilization time (GST), after which message delays are at most a known upper bound Δ. Before GST, there are no guarantees that messages will be delivered and can be lost. This is the common system model adopted by Byzantine consensus protocols since it captures real-world scenarios where network delays fluctuate.

2.1 State Machine Replication

State machine replication refers to a system that provides a replicated service whose state is copied across a set of processes. This redundancy eliminates single points of failure and enhances the reliability of the service. The term is typically used in the context of systems where clients issue commands and processes must agree in the order of execution of these commands. In addition to processes, any number of clients can be also faulty in the respective model.

SMR is defined by the following properties [29]:

- **Initial State**. Every non-faulty process must have the same initial state.
- **Determinism**. Every non-faulty process for a given state and a given input, must produce the same output and transition to the same new state.
- **Coordination**. Every non-faulty process must process commands in the same order.

Typically, in distributed systems, SMR is achieved by sequentially executing multiple instances of consensus, in which processes attempt to agree on the execution order of requests.

2.2 Consensus

Consensus is a fundamental problem for structuring reliable and fault-tolerant distributed services where multiple processes must agree on a single value even in the presence of faults. This problem involves processes proposing their candidate values, communicate with one another, and agree on a single common decision. Consensus is closely related to several fundamental abstractions in distributed systems. For instance, solving consensus and achieving SMR is equivalent to solving total order broadcast, in which processes must deliver the same set of messages in the same order, even in the presence of faults. In practice, many systems implement SMR and total order broadcast in the form of a blockchain, where the agreed sequence of values is organized into blocks that form an append-only chain. The data structure underlying a blockchain is commonly referred to as a ledger, which represents a persistent, immutable log of all committed transactions.

The consensus problem defines three properties that must be satisfied [18]:

- **Validity**. The agreed value must have been proposed and must be valid in the application's context.
- **Agreement**. All non-faulty processes must agree on the same value.
- **Termination**. Every non-faulty process eventually decides on a value.

The goal of consensus protocols is to achieve safety (agreement and validity) and liveness (termination). Consensus is often solved using quorums, which are subsets of processes that represent a majority or a critical portion of the system. Quorums can be used to ensure that processes in a distributed system agree on a decision even in the presence of faults and are the minimal subset of processes in the

system required to make a decision. The quorum size is chosen such that overlapping between quorums guarantees consistency between decisions of different processes.

Probabilistic Consensus. This category of consensus differentiates from the traditional BFT protocols, in the sense it abandons the requirement of ensuring agreement and termination properties in a deterministic way. Specifically, a protocol that solves probabilistic consensus satisfies the following properties [3]:

- **Validity**. The value decided by a non-faulty process satisfies an application-specific predicate.
- **Probabilistic Agreement**. Any two non-faulty processes decide on different values with a certain probability depending on the number of existing Byzantine processes and quorum sizes.
- **Probabilistic Termination**. Every non-faulty process eventually decides with probability 1.

2.3 Blockchain

Blockchain is a term popularized by Satoshi Nakamoto in 2008 with the introduction of Bitcoin [27]. The usage of blockchain extends to many different applications due to its provision of an accurate and trustworthy record and security for decentralized systems. The most well-known application of blockchain is part of the financial industry, specifically, cryptocurrency. Additionally, blockchain's usage extends to the fields of cloud storage, healthcare systems, energy and government.

A blockchain is an ordered log of sequential blocks linked using cryptographic hashes, each containing a batch of transactions that never decreases in length. A block, besides containing a set of transactions, contains a cryptographic hash of its parent block making all blocks interconnected up to first block of the chain, the genesis block. A genesis block is known to all participants in the beginning of a consensus protocol and does not contain any transactions. Once a block is committed to the blockchain, it becomes immutable and cannot be changed, ensuring the integrity and security of the data. At any point, a process's output in a blockchain protocol is its local copy of the blockchain it maintains.

We define blockchain as follows (adapted from [5]). A protocol implements blockchain if it satisfies the following properties:

- Consistency: If two correct players ever output sequences of transactions $\mathcal{B}$ and $\mathcal{B}'$ respectively, either $\mathcal{B} \preceq \mathcal{B}'$ or $\mathcal{B}' \preceq \mathcal{B}$, where "$\preceq$" means "is a prefix of or is equal to."
- Liveness: If a valid input txs is provided to every correct process, then it eventually appears in the output of all correct processes.

2.4 Message and Communication Complexity

Both message and communication complexities are metrics used to evaluate the efficiency of distributed protocols. Message complexity and communication complexity of a consensus protocol refer, respectively, to the number of messages exchanged among processes to reach consensus and the total number of bits transmitted to achieve a decision. These metrics are commonly represented using the Big-O notation, which is used to describe the asymptotic behavior of algorithms and are dependent on the number of processes in the system, n.

2.5 Responsiveness, Finalization and Block Time

Responsiveness refers to a property of a consensus protocol where the time it takes to finalize a decision depends only on the actual message delays and processing times, and not on pre-configured timeout values [9], [12]. A responsive consensus protocol decides as quickly as the network and processes allow. This property is particularly important in consensus protocols because responsiveness protocols present much higher throughput and shorter finalization and block times.

In the literature, δ represents the actual network delay or processing time, while Δ, is an upper bound on δ and represents the maximum network delay in partially synchronous systems. The finalization time is the time that a process in a consensus protocol takes to see the consensus decision. On the other hand, the block time is a concept used in the context of blockchain consensus protocols and represents the time between the addition of successive proposed blocks to the blockchain. Both the finalization time and the block time depend on δ, and/or Δ, depending on the specific protocol.

Chapter 3

Related Work

This section presents overviews of several Byzantine fault-tolerant SMR and blockchain protocols that are fundamental to understanding the evolution of consensus in distributed systems. We begin by presenting a classical leader-based BFT protocol, PBFT [8], which established the groundwork for practical implementations of BFT consensus. Then we present more recent streamlined protocols, Streamlet [10] and HotStuff [35], which propose simpler and more efficient approaches to BFT consensus, respectively. Finally, we highlight evolutions of these protocols (e.g., Simplex [9] and Jolteon [15]) that focus on reducing communication overhead and solving their respective limitations.

A comparison of the presented protocols is depicted in Table 3.1, which denotes the protocol, message complexity, communication complexity, number of communication steps, responsiveness and, block and finalization times for the common case of blockchain protocols. As a side note, for the counting of communication steps, client-server communication was not considered.

3.1 PBFT

Practical Byzantine Fault Tolerance (PBFT) [8] is an SMR protocol introduced by Miguel Castro and Barbara Liskov in 1999. PBFT was the first practical implementation of Byzantine consensus making it the foundational protocol that has influenced most modern Byzantine fault-tolerant protocols designed for distributed systems like blockchain and decentralized networks (e.g. [6], [35], [22]).

PBFT assumes a partially synchronous distributed system and considers a system with a total of $3f + 1$ processes that tolerates up to f Byzantine faults and digital signatures of messages.

PBFT follows a leader-based approach and makes progress by changing views. In PBFT, a view is considered as the period during which a designated leader is responsible for coordinating the consensus

Table 3.1: Comparison of BFT consensus and blockchain protocols in the *common case*.

Protocol	Msg. Complexity	Comm. Complexity	Block Time	Finalization Time	Responsive
PBFT	$O(n^2)$	$O(n^2)$	–	3δ	✓
Streamlet	$O(n^3)$	$O(n^3)$	2Δ	6Δ	×
MinStreamlet+	$O(n^2)$	$O(n^2)$	2δ	4δ	✓
Simplex	$O(n^2)$	$O(n^2 \times \textit{blockchain size})$	2δ	3δ	✓
DispersedSimplex	$O(n^2)$	$O(n^2)$	2δ	3δ	✓
HotStuff	$O(n)$	$O(n)$	2δ	7δ	✓
Jolteon	$O(n)$	$O(n)$	2δ	5δ	✓
ProBFT	$O(n\sqrt{n})$	$O(n\sqrt{n})$	–	3δ	✓

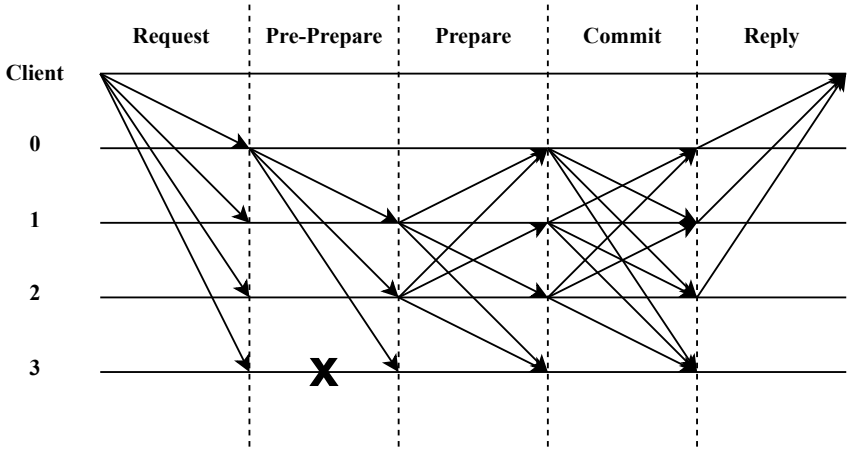

Figure 3.1: PBFT's normal operation (adapted from [8]). Process 0 is the primary and process 3 is faulty.

process. There are two operation modes, normal operation, when the system behaves correctly and view-change, when faulty leader behavior is observed. During each view, a primary process (leader) is responsible for ordering client requests and multicasting these requests to the other backup processes. The normal operation mode of the protocol (illustrated in Figure 3.1) is composed of 5 communication phases:

- **Request**. A client sends a request to all processes.
- **Pre-prepare**. The primary assigns a unique sequence number to the request and sends the request to all backup processes.
- **Prepare**. Each process verifies the validity of the request and sequence number, and sends a *prepare* message to all processes to avoid being deceived by a Byzantine leader.
- **Commit**. After agreeing on the sequence number of the request and after receiving prepare messages from a quorum of processes, each process sends a *commit* message to all other processes to agree on the request.
- **Reply**. When processes agree on the request, after receiving commit messages from a quorum of processes, each one sends a reply to the client. When a client receives $f + 1$ reply messages, it considers the request executed.

The PBFT protocol has a message complexity of $O(n^2)$ since, in the normal operation mode, each process sends messages to all other processes. Despite this, the protocol's communication complexity is $O(n^3)$ since during a view change, when the leader is suspected of being faulty, the new leader must collect *view-change* messages from all processes, each containing a certificate (i.e., a set composed by at least $2f + 1$ signatures). These certificates guarantee if a value has been committed or prepared before. The new leader then broadcasts a *new-view* message, containing the collected certificates, to all processes to initiate the next view. At this point, if the new leader does not propose the value prepared or commited before, correct processes do not consider the proposal valid.

During the years, several works looking to improve the PBFT emerged. For instance, Zyzzyva [21] improves PBFT's performance by implementing a speculative execution, where each process speculatively executes a request just after receiving its sequence number from the primary. This work reduces

PBFT's latency to a total of 3 communications steps since it considers that the consistent state of the processes only matters to clients, who will verify if all processes are on the same state. Spinning [33] increases performance by avoiding Byzantine leaders presence in multiple views and distributing workload among all processes, which is achieved by changing view in every ordering of requests. MinBFT [34] improves PBFT's cost by introducing a trusted service that enables a reduction on the number of processes required for the protocol to tolerate a fixed number of faults. Alternatively, MinBFT also improves PBFT's resilience when considering a fixed number of total processes by tolerating a higher number of faults.

3.2 Streamlet

Streamlet [10] is a streamlined blockchain protocol whose main offering is its minimalist approach to consensus, prioritizing simplicity and ease of implementation. Streamlet is simpler than PBFT because it does not require a complex view-change sub-protocol when faulty leaders are present during the protocol's execution and instead only uses two different types of messages. The authors of Streamlet present three different versions of the protocol, each one associated with a different system model. In the case of this work, the focus will be solely on the Byzantine fault tolerant version for partially synchronous networks which is the same system model assumed by the work we present in this thesis.

The protocol is leader-based and tolerates at most f faults in a system composed by $3f + 1$ processes. All processes have a local synchronized clock responsible for advancing epochs that have a defined duration of 2Δ. In each epoch, the pre-determined leader proposes a new block based on the current state of its local blockchain and afterwards, the remaining processes can vote for the new block. After GST, messages between correct processes arrive at most in Δ time units. Every process signs messages with its private key. Messages are verified using the sender's public key. A vote on a block is a signature performed on the leader's proposal. Each block, despite containing a hash of its parent block and a set of unconfirmed transactions to be executed, also contains the epoch number in which the block was proposed. In Streamlet, when a process collects at least $2f + 1$ votes for a certain block, it becomes *notarized* for that process. Additionally, we refer to the longest notarized chain as the chain of notarized blocks with the greatest length.

During each epoch *e*, the protocol works as follows:

- **Propose**. In the beginning of epoch *e*, the epoch leader creates a new block containing pending transactions that extends the longest notarized chain that it has seen at the time of the proposal. Then, the leader sends the proposal to every process.
- **Vote**. During epoch *e*, each process votes for the first proposal of epoch *e*'s leader if the proposed block extends the longest notarized chain of the process's local blockchain.
- **Finalization**. When a process observes three adjacent blocks in a notarized chain with consecutive epoch numbers, it considers the second of three blocks and its prefix chain finalized.

Figure 3.2 illustrates the finalization rule. After a block is finalized, all transactions contained in the block are confirmed and cannot change. Despite Streamlet's simplicity and minimalist design, it exhibits practical limitations. Mainly, Streamlet requires that every process echos each message received to guarantee the protocol's liveness. These limitations are addressed by other protocols such as Min-Streamlet [2] and Simplex [9] (see Sections 3.3 and 3.4 respectively) and are presented in more detail in the next section.

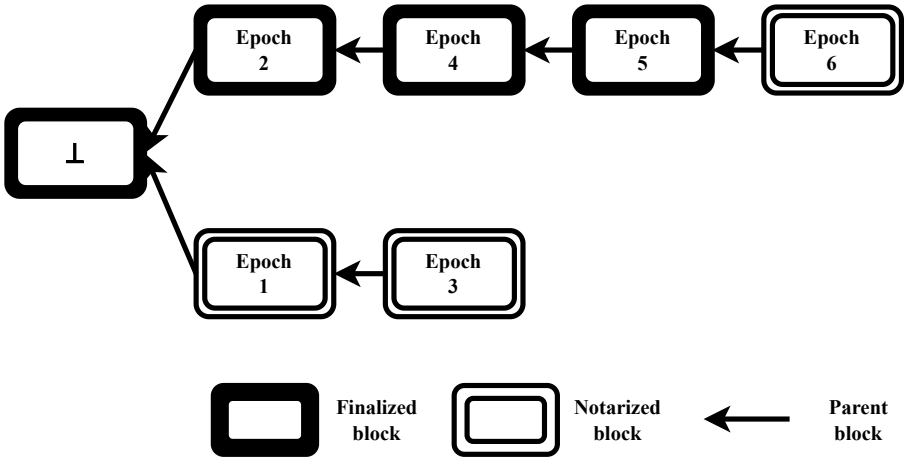

Figure 3.2: Streamlet's finalization rule (adapted from [10]).

3.3 MinStreamlet+

MinStreamlet+ [2] is a variant of Streamlet with improved message complexity, reduced number of steps required to finalize blocks, responsiveness, and increased resilience. In particular, MinStreamlet+ effectively solves the following Streamlet's limitations.

Implicit echo. Streamlet utilizes an implicit echo mechanism in order to guarantee liveness. Specifically, it requires the echoing of every received message so that every process is guaranteed to observe every message exchanged in the protocol. This mechanism implies that a Byzantine process cannot send conflicting messages or omit messages to a subset of replicas without being detected. If the echo mechanism were removed, a Byzantine process could block the progress of a subset of processes by causing their view of the blockchain to diverge from that of others, thereby preventing the voting phase. Despite being essential, the implicit echo mechanism comes at the cost of a cubic message complexity since processes broadcast both their own votes and votes received from other processes. Consequently, Streamlet's communication complexity is also $O(n^3)$ because the size of all messages' types is not dependent on the number of processes of the system.

High latency. Three adjacent blocks with consecutive epochs are required to finalize a block in Streamlet. This means that at least six communication steps are required to finalize a block resulting in a finalization time of 6Δ in the best case-scenario.

Non-responsive. In Streamlet, even if a correct process notarizes a block before the epoch time expires, it must wait for the full epoch time until a new one starts, weakening the protocol's throughput.

To address these limitations, MinStreamlet+ introduces the following optimizations.

Echo removal. MinStreamlet+ replaces the implicit echo mechanism of Streamlet by letting a process that has not seen the previous notarized block initiate a recovery sub-protocol to retrieve missing blocks. This is accomplished by sending a request to $f + 1$ processes, asking them to respond by sending a notarization proof of the missing blocks. This modification reduces the message complexity to quadratic in the base-case scenario, where the leader is correct after GST.

Achieving responsiveness. Responsiveness in Streamlet can be achieved by advancing epochs as soon as a block becomes notarized but requires a mechanism to skip epochs when progress is not made. This is achieved by integrating a timeout mechanism. If a process does not see a proposal or a new notarized block in the current epoch, it broadcasts a timeout message with the epoch it wishes to advance to. A process will advance to a new epoch when it sees $n - f$ timeout messages or a new notarized block. This modification improves Streamlet's block time from 2Δ to 2δ.

Improving latency. In Streamlet, if there are multiple notarized chains of the same length, the leader can randomly extend one of them. MinStreamlet+ applies a concept of chain freshness which allows breaking ties between chains of the same length by choosing the most recent one. This modification allows the finalization of the first of the two adjacent blocks with consecutive epochs and reduces the latency to finalize a block to 4δ.

Improving resilience. MinStreamlet+ applies a trusted service component that enables correct processes to detect equivocation caused by Byzantine processes. This change can enhance the system's resilience by reducing the number of processes required to tolerate f Byzantine faults from $3f + 1$ to $2f + 1$.

3.4 Simplex

Simplex [9] is another protocol which improves Streamlet while also aiming for simplicity, and at the same time achieves a faster finalization time than all competing protocols. Simplex has a very similar model to Streamlet. It is a leader-based protocol that follows the propose/vote approach, tolerates at most f faults in a system composed by $3f + 1$ processes, executes under a partially synchronous network and uses a public-key infrastructure to sign and verify messages where every process has a public key known to every other process and a private key. Distinctly, Simplex is a responsive protocol and does not have any of the limitations of Streamlet presented in Section 3.3.

Simplex achieves responsiveness by advancing epochs only when blocks are notarized (depicted in Figure 3.3). Processes are equipped with a timer that expires after 3Δ if a notarization for a block of length *h* does not happen during iteration *h* (same concept as epoch in Streamlet). Simplex uses dummy blocks that may be proposed when the timer expires (the network is slow or the leader is faulty) and voted and agreed on, only if a finalization for iteration *h* has not happened yet. The finalization of transactions is also different from Streamlet and achieves better latency. In Simplex, after a notarization of a block of length *h* during iteration *h*, processes multicast a *finalize* message for iteration *h* if their timer has not expired during the corresponding iteration. At any point, if a process sees *2f + 1* signed *finalize* messages, it considers the blocks notarized in iteration *h* and all previous iterations finalized. Regardless of whether the process sent a *finalize* message, if it notarized a block of length *h*, it multicasts a *view* message containing its current view of the notarized blockchain to every other process, to ensure that other processes will also enter iteration $h + 1$ within one message delay.

The Simplex protocol presents a block time of 2δ since only the propose and vote phases are necessary for a block notarization to advance the current iteration and cause the next leader to propose a new block. On the other hand, the protocol has a finalization time of 3δ due to the additional finalization phase. Additionally, Simplex achieves a message complexity of $O(n^2)$ to finalize a block in the best-case scenario since all processes broadcast their messages during the vote and finalize phases to every other process and, a communication complexity of $\mathcal{O}(n^2 \times \textit{size of the notarized blockchain})$ because Simplex's *propose* and *view* messages must include the entirety of the process's local notarized blockchain.

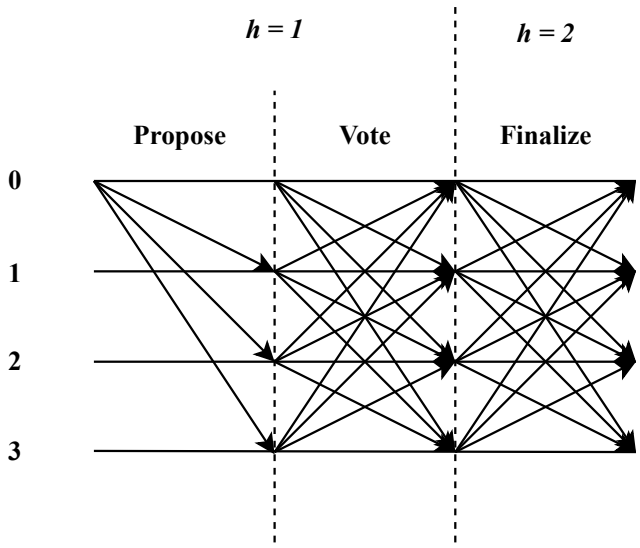

Figure 3.3: Simplex's block notarization. Process 0 is the leader.

Despite Simplex's achievements, it still presents the following practical limitations which we address and present a solution in this work (see Chapter 4).

High communication complexity. Simplex's *propose* and *view* messages must include the entirety of a process's local notarized blockchain which increases the communication complexity of the protocol and also floods the network with bigger and bigger messages as the protocol continues making progress.

Increased blockchain length. Adding dummy blocks to the blockchain can come as an additional memory cost that may not be feasible or justified since these blocks do not include any significant information (transactions).

3.5 DispersedSimplex

A variation of the Simplex protocol called DispersedSimplex [30] was recently published. DispersedSimplex fleshes out some crucial limitations of Simplex with the objective of obtaining a protocol with acceptable communication complexity. Just like Simplex, DispersedSimplex is a leader-based protocol in which leaders are rotated in a round-robin fashion. Similarly, the protocol iterates over a sequence of slots (iteration in Simplex parlance) where for each slot, there is a designated leader responsible for proposing a new block. In the same manner, the system tolerates up to f Byzantine faults and is composed by $3f + 1$ parties (processes), under the assumption of a partially synchronous network and utilizing a public-key infrastructure to sign and verify incoming messages.

The modifications of DispersedSimplex allow leaders to disperse blocks in a more communication-efficient way by making use of well-known techniques involving erasure codes and Merkle trees. It introduces the concepts of support, commit and complaint shares. A support share is an object holding a valid signature on a proposed block and a certified fragment for the same block. Certified fragments result from the encoding operation of a given payload and are used to reconstruct messages. Commit and

complaint shares are both a valid signature on a object containing a slot number v. Support, commit and complaint certificates are a collection of $2f + 1$ support, commit and complaint shares respectively. Each party maintains a complete block tree rooted at the genesis block at slot 0. There will be at most one block for any given slot in the tree. Each party also maintains a certificate pool. Whenever a party receives a quorum of support, commit, or complaint shares and it does not already have a corresponding certificate, a new certificate will be created and added to the pool. At this point, a correct party broadcasts the certificate to all other parties. A correct party also adds a certificate to its certificate pool and broadcasts it to all other parties, when it receives a support, commit, or complaint certificate, and it does not already have a corresponding certificate. With this strategy, DispersedSimplex improves the communication complexity of Simplex without including the entirety of the local blockchain or block tree in messages. In the same way, it allows late parties that may have not observed a proposal for slot v (either because a leader crashed during sending the proposal or a leader is Byzantine) to enter the next slot $v + 1$ when a valid support certificate is received.

The protocol starts with the leader of a slot v sending a block proposal to all parties including v', which is the slot number of the last block added to the block tree. Upon receiving such a message, each party validates the proposal and sends the respective support share to all other parties. The validation of a block proposal includes verifying if the certificate pool contains complaint certificates for slots $v' + 1$, ..., $v - 1$. Each party will move onto the next slot when it adds the proposed block to its complete block tree after observing a quorum of support shares for it. If a timer expires before a block is added to the complete block tree for the current slot (e.g., 2Δ), a party will broadcast a complaint share for v and move onto the next slot when it obtains a corresponding complaint certificate containing a quorum of complaint shares for v received from other parties. A party will only broadcast a commit share for v when it advances to next slot in consequence of adding the proposed block to its block tree and has still not issued a complaint share for v. Just like in Simplex, this rule is essential for safety. Finally, a party considers a block committed and all its predecessors if its block tree contains the supported block and if its certificate pool contains a commit certificate for slot v.

DispersedSimplex's modifications result in a responsive protocol with $O(n^2)$ message complexity and an improved $O(n^2)$ communication complexity.

3.6 HotStuff

HotStuff [35] is another BFT protocol that solves the SMR problem for the partially synchronous network model. HotStuff was designed to simplify and enhance the efficiency of classical BFT protocols (e.g., PBFT, Zyzzyva, BFT-SMaRt [6]). The main contribution of HotStuff is the design of a protocol capable of achieving both responsiveness and linear message complexity for committing a block after GST in the common case. A variant of the protocol, known as Chained HotStuff, adopts a pipelined consensus style. Rather than requiring multiple voting stages per decision, the vote for one decision is piggybacked to the voting phase of the subsequent one, reducing overhead and improving the overall throughput of the consensus process. More recently, HotStuff-2 [26] has been proposed as a refined version of the protocol, aiming to further reduce latency and communication overhead by reducing the number of phases from three to two, while preserving the linearity and responsiveness of HotStuff.

HotStuff considers a system composed by $3f + 1$ replicas and a threshold-signature scheme in which there is a single public key known by all replicas and each replica holds its unique private key. The protocol works in a succession of views and for each one, there is a dedicated leader known to all replicas. For a decision to occur, the leader of a particular view must collect votes from a quorum of

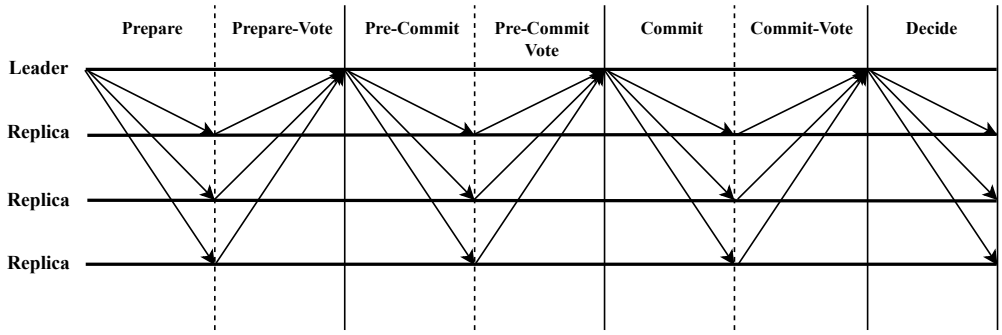

Figure 3.4: Representation of HotStuff's execution.

$n - f$ replicas in three phases, prepare, pre-commit, and commit:

- **Prepare**. The leader starts by collecting $n - f$ *new-view* messages from other replicas. These messages are sent by a replica transitioning into a new view and include the highest *prepare* quorum certificate (*prepareQC*) that the replica has seen. From these messages, the leader selects the *prepareQC* with the highest view. Then it sends a *prepare* message with a proposal extending the selected *prepareQC*. Upon receiving a *prepare* message for the current view from the leader, a replica validates the proposal and sends a *prepare-vote* containing a signature to the leader.
- **Pre-commit**. When the leader receives $n - f$ *prepare-vote* messages for the current proposal, it combines them into a *prepareQC*. The leader then broadcasts a *pre-commit* message containing this *prepareQC*. Afterwards, a replica responds to the leader with a *pre-commit-vote* containing a signed digest of the proposal.
- **Commit**. When the leader receives $n - f$ *pre-commit-vote* messages, it combines them into a *pre-commit* quorum certificate (*precommitQC*) and broadcasts it in *commit* message. Replicas respond to it with a *commit-vote* and become locked on the *precommitQC* at this point. At this point of time, the leader waits for $n - f$ *commit-vote* messages, combines them into a *commit* quorum certificate (*commitQC*) and sends it in *decide* message to all other replicas. Upon receiving a *decide* message, a replica considers the proposal embodied in the *commitQC* a committed decision and starts the next view.

Overall, for a single decision to be committed, HotStuff takes 7 communications steps (see Figure 3.4). This many communication steps results in an increased latency compared with other protocols (e.g. PBFT) and is the main drawback of HotStuff.

3.7 Jolteon

Jolteon [15], is a 2-chain version of the HotStuff protocol. In particular, Jolteon preserves the structure of HotStuff and its linearity under synchrony with correct leaders while reducing the block-commit latency by 30% using a 2-chain commit rule. This decrease in latency comes at the cost of a quadratic view-change. More specifically, the complexity of proposing a block after a bad view that requires synchronization is linear for HotStuff and quadratic for Jolteon. On the other hand, the complexity of view-synchronization under asynchrony or failures is quadratic for both protocols, due to all-to-all

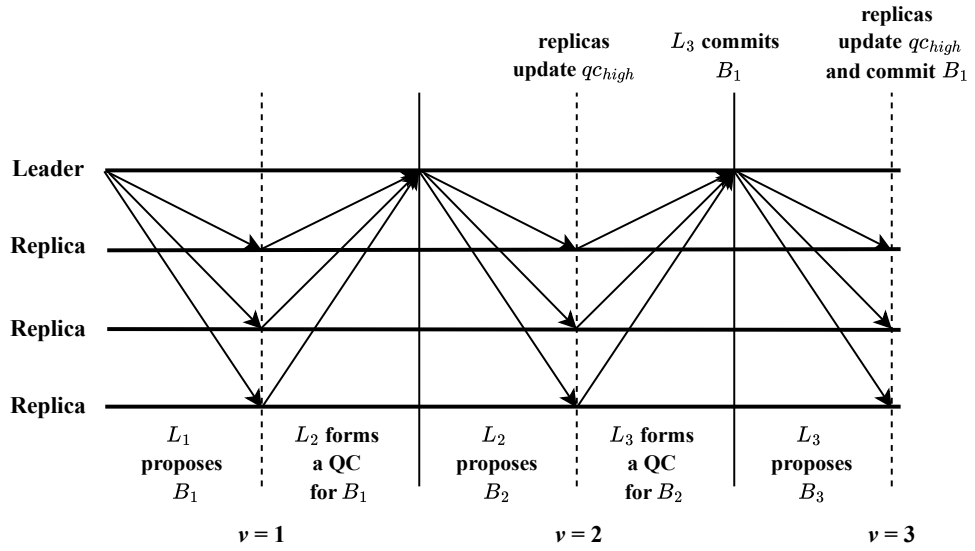

Figure 3.5: Jolteon's 2-chain commit rule.

timeout messages. Jolteon states this trade-off as justifiable because HotStuff's view-change linearity is limited by the the quadratic cost of view-synchronization which is inherent in leader-based consensus protocols.

In Jolteon, a leader is responsible for proving the safety of its proposal. In a common case, just like in HotStuff, each proposal extends the highest quorum certificate selected by the leader and providing this quorum certificate is sufficient to prove safety. Additionally, a proposal must include a *timeout* certificate (TC) used for view-change, containing $2f+1$ highest quorum certificates sent by replicas after a timeout caused by asynchrony or a faulty leader. This proposal sent to all other replicas possibly containing a TC is the cause of quadratic view-change communication in Jolteon ($O(n)$ messages of $O(n)$ size). This change is necessary to prove that a more recent or higher quorum certificate of a commited block does not exist. When a replica receives a proposal, before sending a vote to the leader of the next view, it verifies if it contains the quorum certificate for the block of the previous view or if it contains the highest quorum certificate among the $2f+1$ quorum certificates in the TC. If this verification succeeds, a replica advances its view number, updates its highest quorum certificate and checks the 2-chain commit rule. In the case of Jolteon, the 2-chain commit rule states that whenever two adjacent blocks that received enough votes with consecutive view numbers exist in the chain, the replica commits the first of the two blocks and its ancestors. The 2-chain commit rule reduces the number of communication steps for a decision to be made from 7 in HotStuff, to 5 in Jolteon (see Figure 3.5).

3.8 ProBFT

ProBFT [3] is a BFT probabilistic consensus protocol. Traditionally, Byzantine consensus protocols assume an extremely pessimistic system model when dealing with Byzantine adversaries to ensure safety and liveness primitives even under worst-case scenarios, however in practice, adversaries are not usually as powerful as the ones assumed. Indeed, in many real-world applications, it is sufficient to assume a

static corruption adversary, which chooses the set of faulty replicas at the beginning of the execution of a consensus instance [3]. ProBFT aims at less pessimistic practical scenarios where, ensuring deterministic quorum overlaps can pose inherent challenges in achieving both resource efficiency and high performance and less message exchanges and optimal latency are required. In these less pessimistic practical scenarios, it may suffice to ensure safety and liveness properties with a high probability. ProBFT aims at quorums intersecting with high probability, allowing the number of communication steps to be at a minimum and the reduction of quorum sizes. This approach obtains improved resource consumption and scalability.

We provide a detailed presentation of the ProBFT protocol, along with an outline of its correctness proofs, because its core mechanisms constitute the primary inspiration for the protocols developed in this thesis and are directly incorporated into our work.

3.8.1 System Model

The protocol considers a distributed system composed by a finite set of n replicas and at most $f < n/3$ Byzantine failures. The set of Byzantine faulty replicas is fixed at the beginning of the execution of the protocol and is unknown to the other correct replicas (static corruption adversary). Furthermore, ProBFT assumes an adversarial scheduler that manipulates the delivery time of messages independent of the sender's identifier and if it is faulty or not. Each replica has a unique ID and an associated private key that it uses to signs outgoing messages and only accepts an incoming message if the message's signature can be verified using the sender's public key. The private key of all correct replicas is considered to remain secret and unknown to faulty replicas. Additionally, an adversary cannot break cryptographic primitives. The protocol assumes a partial synchronous network model in which the system acts asynchronously and messages can be lost until an unknown GST. After GST, the system becomes synchronous and there is a unknown time limit for communication and computation.

3.8.2 Protocol

ProBFT operates in a sequence of views and each one has a designated leader responsible for proposing a value. In ProBFT, views are produced by a synchronizer [7] responsible for notifying replicas when a view changes. Each view consists of three communication steps (depicted in Figure 3.6):

- **Propose**. The leader of view *v* proposes a value by broadcasting a *propose* message to all other replicas.
- **Prepare**. Upon receiving a *propose* message, a correct replica multicasts a *prepare* to a sample of $o \times q$ distinct replicas chosen randomly from the set of *n* replicas, where $q = l\sqrt{n}$, $o > 1$ is a real constant and *l* is a configurable, typically small constant.
- **Commit**. After a correct replica receives *prepare* messages from a probabilistic quorum with size *q*, it multicasts a commit message to another random sample of $o \times q$ distinct replicas. When it receives *commit* messages from a probabilistic quorums of size *q*, the replica decides on the value.

To offer resilience against faulty replicas capable of manipulating decisions in probabilistic quorums, ProBFT delegates the receptors of messages to a globally known verifiable random function (VRF). This VRF provides two operations:

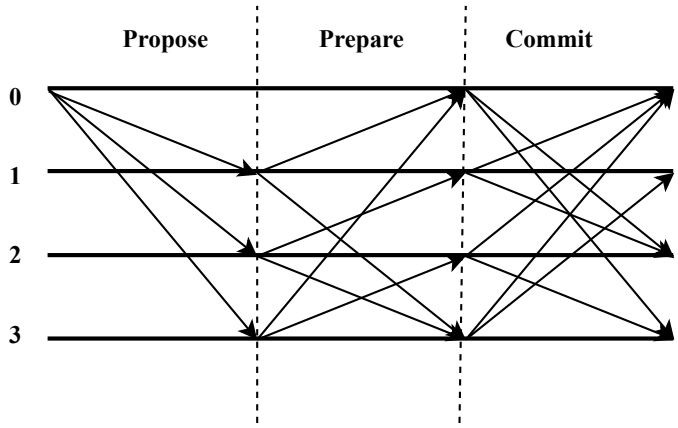

Figure 3.6: Representation of ProBFT's execution (adapted from [3]). Replica 0 is the leader.

- **VRF_prove**. Given a replica *i*, its private key, and a seed *z*, VRF_prove returns a sample S_i with the IDs of the different replicas selected at random and a proof P_i that allows other replicas to verify whether the sample was obtained using this operation.
- **VRF_verify**. Given a public key of a replica *i*, a seed *z*, a positive integer *s*, a sample S_i and its proof P_i, VRF_verify returns a boolean indicating if S_i was obtained using VRF_prove with the given parameters.

3.8.3 Message and Communication Complexity

ProBFT's message complexity is $O(n\sqrt{n})$ since each correct replica, after receiving a *propose* or *prepare* message, multicasts another message to a sample of replicas of size $o \times l\sqrt{n}$. For any view greater than one, a correct replica sends a *newLeader* message to the leader of the current view. This new leader then sends a *propose* message with a certificate containing a full (not probabilistic) quorum of *newLeader* messages to all processes. Each *newLeader* message might contain a certificate with a probabilistic quorum of *prepare* messages. Hence, ProBFT's communication complexity is $O(n^2\sqrt{n})$.

3.8.4 Outline of Correctness Proofs

In the following, an outline of the correctness proofs of ProBFT is presented. In particular, we summarize the proofs for validity, probabilistic termination and probabilistic agreement presented in [4].

Validity. In ProBFT, after receiving a proposal from the leader of a view v, a correct replica verifies its validity. One of the conditions necessary for a valid proposal is that the proposed value satisfies the protocol's `valid` predicate. Consequently, a value decided by a correct replica is valid, satisfying the Validity property.

Probabilistic termination. In a scenario where a subset of replicas has still not decided a value by GST, there will be a correct replica from this subset that performs the leader's role and proposes a value x in view v . All replicas receive such a proposal during view v since the system is synchronous after GST and multicast their *prepare* messages. At this point there is a high probability that a correct replica will

decide the value x. Even in the event that a correct replica might not receive enough messages to form a quorum that allows it to decide x, there are infinite views whose leaders are correct, leading to each correct replica deciding with probability 1.

Probabilistic agreement. Different replicas can decide different values in ProBFT since quorum intersections are not guaranteed in a deterministic way. The probability of agreement violation is low and such a scenario only occurs when a leader is Byzantine and proposes multiple values. For a detailed analysis on the exact probability of agreement violation for different scenarios refer to [4].

Chapter 4

Overcoming Simplex Practical Limitations

This chapter presents an adaptation of the Simplex protocol [9] to address the practical limitations identified in Section 3.4. For the remainder of this dissertation, this adaptation will be referred to as Practical Simplex. As previously mentioned, Simplex presents a crucial limitation that prevents it from realistically being used in practical scenarios. In each iteration all processes send a complete copy of the blockchain to every other process, along with a certificate containing the vote signatures for each block from the genesis block up to the current iteration. This is entirely impractical and has also been addressed by DispersedSimplex [30].

The primary objective of Practical Simplex is twofold. First, it serves as a foundation for the integration of ProBFT's [3] mechanisms, ultimately giving rise to the probabilistic chained blockchain protocol ProSimplex (introduced in Chapter 5). Secondly, it solves the practical limitations of the original Simplex design.

We introduce the optimizations present in Practical Simplex, followed by the protocol description and outline of the correctness proofs. In the end, we compare Practical Simplex achievements with DispersedSimplex.

4.1 Optimizations

Improving communication complexity. Before presenting the solutions to the practical limitations of Simplex, we demonstrate how removing complete copies of the notarized blockchain from *propose* and *view* messages stops the protocol from making progress, leading to a liveness issue. Recall that messages can be lost or not received until GST. If the local notarized blockchain would to be removed from these types of messages, a possible scenario would be a process's notarized blockchain lagging behind because it did not receive any of the messages exchanged before GST. Besides, even after GST, there would be no way for a process to synchronize its local state. A representation of this situation is depicted in Figure 4.1. Consider the presented example where, during iteration $h = 1$, all processes besides process 3 observe a quorum of *vote* messages ($2f + 1$ votes) for a proposed block b before GST. In iteration $h = 2$, after GST, all processes receives enough *vote* messages for a proposed block b', however process 3 does not vote for block b' since it is still in iteration $h = 1$ because it never notarized block b. At this point, process 3 can no longer update its current view of the local notarized blockchain and in the following iterations will not be able to participate in the protocol. In the case of iteration $h = 3$, if process 3 is the leader of some iteration, no other processes will vote for its proposal. Besides, process 3 will no longer vote for the following proposals. After iteration $h = 2$, if process 0 is Byzantine and stays silent for the remaining of the protocol's execution, no correct processes will be able to notarize a new block again,

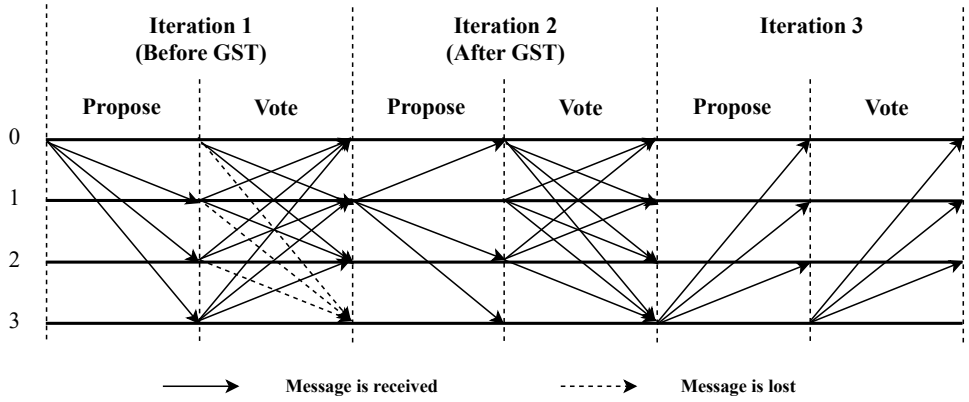

Figure 4.1: Demonstration of Simplex's liveness problem after the removal of view messages.

resulting in a liveness problem. To circumvent this issue, in Practical Simplex, *view* messages (now referred to as *state* messages) were changed to only include the last notarized block of the local notarized blockchain along with its quorum certificate. At the same time, *propose* messages were changed to no longer include a copy of the local notarized blockchain as it turns out to be unnecessary. Additionally, Practical Simplex allows a process to detect that it is lagging behind in the protocol's execution and missing blocks for iterations higher than its current iteration number, when it observes a notarization proof for a block of a higher iteration that it is not able to add to its view of the notarized blockchain. This detection occurs when a process observes at least $2f+1$ correctly signed *vote* or *finalize* messages or, one *state* message containing a notarized block alongside its corresponding quorum certificate containing $2f+1$ correctly signed votes. At this stage, the process sends a *request* message to the sender of one these messages, specifying the current length of its notarized blockchain. When a correct process receives such a *request* message, it responds with a *reply* message, containing the requested missing blocks alongside their corresponding quorum certificates to be verified by the request sender. If the verification succeeds, the late process will update its current view of the notarized blockchain and advance to the most recent iteration. Although, this change increases the number of messages necessary for processes to update their local notarized blockchain when they are lagging behind, it reduces the communication complexity of Simplex to $O(n^2)$ per block, in the best-case scenario.

Timeout mechanism. Taking inspiration from MinStreamlet+'s [2] improvements over Streamlet [10] presented in Section 3.3, Practical Simplex utilizes a timeout mechanism to replace dummy blocks and avoid the additional memory cost originated from these blocks that do not include any significant information. If before the timer for iteration h expires, a notarization for a block does not occur, a correct process broadcasts a *timeout* message with the iteration number $h+1$ that it wishes to advance to. When a process observes $2f+1$ signed *timeout* messages for the same iteration from distinct processes, it advances to the desired iteration. This mechanism does not sacrifice either liveness or responsiveness and maintains the number of messages required for a process to advance an iteration when progress is not made. However, just like in Simplex, this change still allows for forks in the blockchain to exist only after a timeout occurs. Specifically, it is possible for a subset of processes to observe $2f+1$ correctly signed and unique *timeout* messages and a second subset of processes to observe $2f+1$ distinct and valid *vote* messages for the same iteration (see Figure 4.2). Simplex handles this scenario by including the entirety of the local notarized blockchain in *propose* and *view* messages. For example, in Simplex,

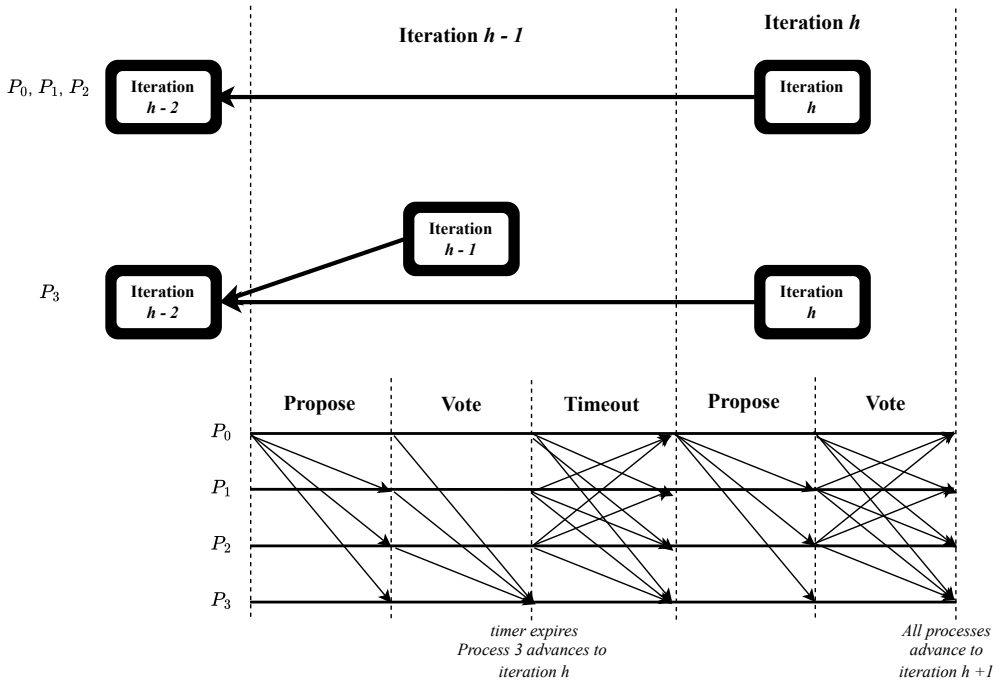

Figure 4.2: Example of Practical Simplex's fork after a timeout.

if a process P_i advances to iteration h via notarizing a dummy block for iteration $h-1$, and another process P_j advances to iteration h via observing a quorum of votes for a block proposal, P_i will replace the dummy block of iteration $h-1$ for the block notarized by P_j, after receiving the first of either a *propose* or *view* message from P_j, containing a valid notarized blockchain of length $h-1$. The Practical Simplex protocol allows for two notarized blocks with the same length to exist in the blockchain and guarantees that only one of the two blocks and its parent chain will be finalized. Consider a scenario where a process P_i advances to iteration h by observing a quorum of *timeout* messages for iteration h, while another process P_j advances to the same iteration via observing a quorum of votes for a block proposal b, which extends a block b' that was notarized by all processes during iteration $h-2$. During iteration h, if P_i is selected as leader and proposes a new block b'' extending b', this block will have the same length as block b. Consequently, all processes that advanced to iteration h via *timeout* messages will vote for b''. At this point, if enough processes vote for b'' and notarize it, P_i may incorporate block b into its local notarized blockchain if it receives a *state* message sent by P_j, after P_j notarizes b in iteration $h-1$, and a corresponding *reply* message containing block b along with its quorum certificate.

4.2 System Model and Preliminaries

Practical Simplex assumes a distributed system composed of $n = 3f + 1$ processes and at most $f < n/3$ processes are faulty. We assume a partially synchronous network model. Under good network conditions (after GST), every message is delivered within Δ time. The network is δ-synchronous at time T if every message sent from a correct process P to another correct process Q, at time T, is received by Q before time $T + \delta$. The network is δ-synchronous over an interval $[a, b + \delta]$ if it is δ-synchronous at time T

for all $T \in [a, b + \delta]$. Before GST, there are no guarantees on message delivery. Practical Simplex considers a public-key infrastructure. Every process has a corresponding public-key and private-key. Each correct process keeps its private key secret, and each process's public key is known to every other process before the protocol's execution. Any message or payload m signed by a process i, is denoted by $\langle m \rangle_i$ and the digital signature on m is verifiable using process i's public key. Additionally, we assume the existence of a cryptographically secure collision-resistant hash function H.

The Practical Simplex protocol considers the following object data structures:

- **Blocks and Block headers.** A block $b_k = \langle \langle H(b_{k-1}.header), k, h \rangle, txs \rangle$ is composed by a set of transactions, txs, and a block header $b_k.header = \langle H(b_{k-1}.header), k, h, H(txs) \rangle$ where k is the block's length, h is the block's iteration, $H(b_{k-1}.header)$ is the hash of the block's parent header and $H(txs)$ is a hash of the block's transactions.

- **Notarized Block.** A notarized block is a block $b_k = \langle \langle H(b_{k-1}.header), k, h, H(txs) \rangle, txs, cert \rangle$ where $cert$ is a quorum certificate of $2f + 1$ signed *vote* messages from unique processes $\subseteq \{0, ..., n - 1\}$.

- **Blockchain.** A blockchain $\mathcal{B} = \{\langle \langle H(b_{k-1}.header), k, h, H(txs) \rangle, txs, cert \rangle\}_{k=0}^{|\mathcal{B}|}$ is a chained sequence of notarized blocks where each block is linked to its predecessor/parent by the hash of its parent's block header. The blockchain's length is represented as $|\mathcal{B}|$.

4.3 Protocol Specification

In this section, we provide a detailed specification of the Practical Simplex protocol. For completeness, the protocol's pseudocode is presented in Appendix A (Algorithm 2).

Message Processing. Upon receiving a signed payload within a message m or a signed message from a process i, $\langle m \rangle_i$, a process only processes m if and only if $\langle m \rangle_i$ is verifiable using i's public key.

Leader Election. The protocol's leader election is performed in a random fashion. For each iteration h, a predetermined leader L_h is chosen by applying the formula $L_h = H(h) \bmod n$.

Notarization and Advancing Iterations. At the beginning of each iteration h, each process starts a timer. If the timer for an iteration h expires before a notarization for h occurs, a process broadcasts a *timeout* message expressing its wish to advance to iteration $h + 1$. Upon receiving at least $2f + 1$ *vote* messages for a block proposal b during iteration h, and the *propose* message for b, a process considers b notarized by adding b to its notarized blockchain and advances to iteration $h + 1$. Alternatively, if a process receives at least $2f + 1$ *timeout* messages for iteration $h + 1$, it also advances to iteration $h + 1$.

Protocol. For each iteration h, the Practical Simplex protocol works as follows:

- **Propose.** When the leader of iteration h, L_h starts iteration h, L_h creates a block proposal containing a new batch of pending transactions extending the last notarized block of its notarized blockchain. Then, L_h broadcasts a proposal of the form $\langle \text{PROPOSE}, \langle \langle H(b_{|\mathcal{B}|}.header), |\mathcal{B}| + 1, h, H(txs) \rangle, txs \rangle \rangle_i$ to every other process.
- **Vote.** During iteration h, upon receiving the first proposal b from L_h, a correct process votes for b if it extends the last notarized block of its notarized blockchain and if its timer for iteration h has still not expired. A process i votes for a block proposal b by sending a *vote* message of the

form $\langle \text{VOTE}, \langle b.header \rangle_i \rangle$, where $\langle b.header \rangle_i$ represents a signature over b's header. When a process i observes a notarization for iteration h, it broadcasts a *finalize* message $\langle \text{FINALIZE}, h \rangle_i$ if the timer for iteration h has not expired. Additionally, process i also broadcasts a *state* message $\langle \text{STATE}, b.header, cert \rangle_i$ containing the header of the last block notarized during iteration h, and its corresponding quorum certificate, $cert$.

- **Finalize.** During iteration h, upon receiving $2f + 1$ *finalize* messages for iteration h, if a process has observed a notarization for h, it considers the notarized block of iteration h and its parent chain finalized. Otherwise, if the process has not observed a notarization for iteration h, it initiates a state recovery mechanism by sending a *request* message, $\langle \text{REQUEST}, |\mathcal{B}| \rangle_i$, to one of the $2f + 1$ processes that sent a *finalize* message for h.
- **State Recovery.** If some process i receives a *state* message from another process j of the form $\langle \text{STATE}, header, cert \rangle_i$, and it contains a valid quorum certificate for a notarized block that satisfies the condition $|\mathcal{B}| < header.k \vee (|\mathcal{B}| = header.k \wedge b_{|\mathcal{B}|}.header.h \neq header.h)$, it initiates a recovery mechanism by sending a *request* to process j of the form $\langle \text{REQUEST}, l \rangle_i$, where $l = |\mathcal{B}|$. If process j is correct, it replies with a *reply* message $\langle \text{REPLY}, \{\langle b_k.header, b_k.cert, b_k.txs \rangle\}_{k=l}^{|\mathcal{B}|} \rangle_j$ to process i, containing the set of notarized blocks missing from process i's notarized blockchain along with their corresponding transactions and quorum certificates. Upon receiving such a message, process i verifies each quorum certificate in the reply. If the verification is successful, it proceeds to notarize all blocks contained in the *reply* message.

4.4 Message and Communication Complexity

Practical Simplex maintains the same quadratic message communication as Simplex. Specifically, processes exchange $n + n^2 + n^2$ messages (n propose messages sent by the leader + n^2 vote messages + n^2 finalize messages) resulting in $O(n^2)$ message complexity. As previously explained in Section 4.1 the Practical Simplex protocol improves the communication complexity of Simplex to $O(n^2)$.

4.5 Outline of Correctness Proofs

4.5.1 Agreement

Lemma 1. If some correct process observes that a block b is notarized for length k, then no correct process will ever observe another block b' notarized for the same length k, unless at least $2f+1$ processes send a *timeout* message during the iteration in which b was notarized.

Proof. Consider two blocks b and b' with the same length, i.e., $b.k = b'.k$. Each correct process votes for at most one of these blocks — either b or b'. A faulty process, however, may vote for both. Since there are at least $2n/3$ correct processes and fewer than $n/3$ faulty processes, the total number of votes cast across both blocks is strictly less than $2n/3 + n/3 + n/3 = 4n/3$. Recall that to notarize a block, at least $2n/3$ processes must vote for that block. If both b and b' were notarized, their combined votes would be at least $4n/3$, which contradicts the upper bound. Therefore, it is impossible for both b and b' to be notarized at the same length. This guarantees that if a correct process observes a notarization for b at length k, no correct process can observe a notarization for a different block b' at the same length. □

Lemma 2. If two correct processes i and j finalize blocks b and b', respectively, at the same length k, then $b = b'$.

Proof. Consider two correct processes i and j, and a length k. Besides, assume that b and b' are two finalized blocks with same length k. By Lemma 1, these processes cannot observe notarizations for these blocks if a timeout does not occur. Hence, assume a timeout occurs, and b and b' become notarized at the same length. Without loss of generality, assume $b.h < b'.h$, hence, a timeout occurs during $b.h$ for at least $f + 1$ correct processes. By line 4 of Algorithm 2, those $f + 1$ correct processes do not send *finalize* messages for $b.h$, thus, b cannot be finalized, unless another block b'' that extends b becomes finalized. Hence, if b does not become finalized, we have a contradiction, as we assumed b is finalized. If b becomes finalized via finalizing another block b'' that extends b, as $b''.k > b'.k$, b' cannot be finalized, again contradicting our assumption. Therefore it is impossible for i and j to finalize different blocks at length k even when a timeout occurs, which concludes the proof. □

Theorem 1 (Consistency)**.** Let $\mathcal{B}$ and $\mathcal{B}'$ be the sequences of finalized blocks by two correct processes. Then either $\mathcal{B} \preceq \mathcal{B}'$ or $\mathcal{B}' \preceq \mathcal{B}$.

Proof. Let $|\mathcal{B}| = k$ and $|\mathcal{B}'| = k'$, and let $\ell = \min(k, k')$. By Lemma 2, the finalized blocks at each length $1 \leq m \leq \ell$ must be identical. Thus, one sequence is a prefix of the other. □

4.5.2 Liveness

Lemma 3. (Synchronous network and correct leader) For an iteration $h \geq 1$, suppose the leader for iteration h is a correct process i which proposes a block b. Suppose that the first correct process j to enter the loop iteration for iteration h does so at time T, after GST. Further suppose that $\Delta_{timeout} \geq 3\delta$. Then each correct process will finish the loop iteration before time $T + 5\delta$ by notarizing i's proposed block. Moreover, each correct process will eventually finalize b, and this will happen before time $T+6\delta$.

Proof. By time $T + 3\delta$, each correct process will enter iteration h, having either notarized a block for iteration $h - 1$ or observed a quorum of *timeout* messages for iteration h. So, at most by time $T + 3\delta$, the leader i will propose a block b that extends a block b' proposed during h', $h' < h$. Therefore, by time $T + 4\delta$, each correct process will receive i's proposal and will vote for it. By time $T + 5\delta$ each correct process will have notarized block b and advanced to iteration $h + 1$. By the assumption that $\Delta_{timeout} \geq 3\delta$, when each correct process notarizes b, the timeout condition will not be met, and therefore, each correct process will broadcast a *finalize* message for iteration h at this time. Eventually, all correct processes will observe a finalization quorum for iteration h before time $T + 6\delta$. □

Lemma 4. (Faulty leader) For an iteration $h \geq 1$, suppose the leader for iteration h is faulty and that at time T, $T \geq$ GST, some correct process is in iteration h and all other correct processes are in iteration h or previous. Then before time $T + \Delta_{timeout} + 4\delta$, all correct processes finish iteration h.

Proof. All correct processes in an iteration lower than h will enter iteration h at most by $T + 3\delta$, having either notarized a block for iteration $h - 1$ or observed a quorum of *timeout* messages for iteration h. By time $T + 3\delta + \Delta_{timeout}$, every correct process will have either notarized a block for iteration h and/or broadcast a *timeout* message for iteration $h + 1$. In either case, less than δ time units later all correct processes will have advanced to iteration $h + 1$. □

Theorem 2 (Liveness)**.** For an iteration $h \geq 1$, let $\Delta_{timeout} \geq 3\delta$, f be the number of consecutive faulty leaders since iteration h, and T be the time at which the first correct process enters iteration h, for $T >$ GST. After GST, some block is finalized by all correct processes at most by time $T + f(\Delta_{timeout} + 4\delta) + 6\delta$.

Proof. If the leader of iteration h is correct, Lemma 3 implies that its block is finalized by time $T+6\delta$. If the leader is faulty, then by Lemma 4 all correct processes advance to iteration $h+1$ within $\Delta_{timeout}+4\delta$. Repeating this argument, after at most f faulty leaders all correct processes reach an iteration with a correct leader. By Lemma 3, the block proposed in that iteration is finalized within an additional 6δ. Thus some block is finalized by all correct processes no later than $T + f(\Delta_{timeout} + 4\delta) + 6\delta$. □

4.6 Comparison with DispersedSimplex

Both the Practical Simplex and the DispersedSimplex protocols improve Simplex's communication complexity to $O(n^2)$. The main distinction between Practical Simplex and DispersedSimplex lies in how both protocols guarantee liveness in the presence of a Byzantine leader or in the event of a leader crash during proposal dissemination. DispersedSimplex ensures liveness by ensuring correct processes broadcast a signature certificate to all processes whenever a quorum of support, commit or complaint shares is received and no corresponding certificate already exists. This strategy allows late processes to recover quickly since they are able to add a missing block to their block tree within a single message delay, δ, requiring only one communication step to disseminate the new certificate. Despite this, each sent certificate must be accompanied by the corresponding block's information, so that processes which never received the original proposal can reconstruct the block's payload (i.e., transactions) and extend their block tree. While this guarantees fast recovery, it also imposes a significant network overhead, as blocks' information is disseminated every time a new certificate is received or generated and may be redundantly transmitted when no processes where affected by a faulty leader. Meanwhile, in Practical Simplex, the state recovery mechanism guarantees that any process missing one or more blocks' notarizations will eventually notarize all missing blocks by receiving a *reply* message containing the corresponding payloads and quorum certificates of each missing notarized block. The state recovery mechanism requires three communication steps, 3δ, which increases the time needed for late processes to catch up. Nonetheless, Practical Simplex avoids unnecessary network flooding by removing blocks' payloads from *state* messages which are broadcast every time a notarization occurs for some iteration. Instead, *state* messages only contain the quorum certificate of the last notarized block and its corresponding header, which is required to verify the signatures included in the certificate, while the block's payload is transmitted in *reply* messages only sent when explicitly requested by a late process. This reduces the network overhead in the common case, at the cost of slower recovery in the uncommon case where processes may be lagging behind in the protocol's execution.

In the end, DispersedSimplex prioritizes a faster recovery at the expense of increased network usage, while Practical Simplex exchanges recovery speed for communication efficiency by limiting when blocks' payloads are disseminated.

Chapter 5

ProSimplex

The central aim of this thesis is to design a probabilistic chained BFT consensus protocol based on the core mechanisms of ProBFT [3]. The resulting work is ProSimplex, a probabilistic adaptation of the Practical Simplex protocol presented in Chapter 4. ProSimplex implements SMR and achieves both sub-quadratic message and communication complexities, by relaxing the typical pessimistic assumptions of deterministic consensus protocols, as well as optimal good-case latency of 3 communication steps with high probability. Furthermore, ProSimplex is responsive and does not require a view-change sub-protocol like ProBFT, since leader rotation is integrated directly into the protocol's iterative structure. We design ProSimplex by integrating ProBFT's core mechanisms into the Practical Simplex protocol, since Practical Simplex like Simplex [9] provides a straightforward approach for ordering blocks of transactions.

This chapter presents a complete description and pseudocode of ProSimplex, including an outline of the correctness proofs as well as its complexity analysis.

5.1 System Model and Preliminaries

The system model of ProSimplex is similar to that of Practical Simplex, presented in Section 4.2. For the sake of avoiding repetition, we concentrate on the additional assumptions of ProSimplex, in particular verifiable random function and its properties, adversarial model as well as notarized blocks.

A verifiable random function [16] allows a random selection of a subset from a given set which is verifiable and secure. ProSimplex assumes a globally known VRF that provides two operations defined as follows:

- $\mathbf{VRF\_prove}(K_{p,i}, z, s, L) \Rightarrow S_i, P_i$. Given the private key $K_{p,i}$ of a process i, a seed z, a positive integer s, and a process ID, L, `VRF_prove` selects a sample S_i containing $s-1$ distinct processes' IDs at random and L. Along with S_i, this operation returns a proof P_i, enabling other processes to verify whether the sample was obtained using this operation.
- $\mathbf{VRF\_verify}(K_{u,i}, z, s, L, S_i, P_i) \Rightarrow \mathbf{bool}$. Given the public key $K_{u,i}$ of a process i, a seed z, a positive integer s, a process ID L, a sample S_i, and an associated proof P_i, `VRF_verify` determines whether S_i is a valid sample generated using `VRF_prove` with the given parameters. The operation returns `true` if the sample and proof are valid and `false` otherwise.

The VRF must provide the following guarantees [3, 17]:

- **Uniqueness**. A computationally limited adversary must not be able to produce two different proofs P_i and P_i' for the same input parameters $K_{u,i}$, z and s.
- **Collision resistance**. Even when a private key is compromised, it should be infeasible for an adversary to find two distinct seeds z and z' for which the `VRF_prove` returns the same sample.
- **Pseudorandomness**. For an adversarial verifier without knowledge of the proof, the corresponding sample should be indistinguishable from a randomly selected set of process IDs.

ProSimplex distinguishes between two types of quorums, deterministic quorums and probabilistic quorums. Deterministic quorums guarantee intersection in at least one correct process, thereby ensuring consistency across decisions. In contrast, probabilistic quorums intersect only with a certain probability. To achieve sub-quadratic complexity and given the linear nature of the propose phase, ProSimplex employs probabilistic quorums exclusively in the vote and finalize phases. In each of the vote and finalize phases, a correct process must determine a sample of processes using the VRF with size $o \times q$ to whom its messages should be sent , where $o > 1$ is a constant and makes progress after receiving messages from a probabilistic quorum with size $q = l\sqrt{n}$ (l being another constant). The seed used for computing a recipient sample with the VRF is the concatenation $h \| \mathtt{T}$ of the current iteration h, and the type of the corresponding message `T`. A recipient sample always includes the leader process of the next iteration, $h + 1$. The application of deterministic seeds prevents faulty processes from manipulating their recipient samples in order to favor a faulty leader and mislead processes into forming probabilistic quorums for a specific block. Additionally, since a process's recipient sample is computed from its private key, adversaries are not able to predict the individual recipient samples of other processes in advance.

A notarized block in ProSimplex is a block $b_k = \langle \langle H(b_{k-1}.header), k, h, H(txs) \rangle, txs, cert \rangle$ where $cert$ is a probabilistic quorum certificate of q vote messages containing a signature over $b_k.header$, $\langle H(b_{k-1}.header), k, h, H(txs) \rangle$, from unique processes $i \in \{0, ..., n-1\}$, k is the block's length, and h is the block's iteration.

Just like in ProBFT, ProSimplex assumes an adversarial network scheduler that manipulates the delivery time of messages independent of the sender's identifier, its past and current states, and whether it is Byzantine or not.

5.2 Differences between ProSimplex and Practical Simplex

Before presenting the full protocol description of ProSimplex, we enumerate the remaining differences between the ProSimplex and the Practical Simplex protocols.

In ProSimplex, processes may fail to observe a probabilistic quorum of *vote* messages, preventing them from notarizing a block and advancing to the next iteration. To address this, ProSimplex eliminates *state* messages from its specification and instead requires *propose* messages to include both a probabilistic quorum certificate for the last notarized block and the block's iteration. This change is necessary because it avoids increasing the protocol's message complexity to $O(n^2)$. Consequently, any process that did not observe a probabilistic quorum of *vote* messages in iteration h can still advance to iteration $h+1$ upon receiving a *propose* message from the leader of iteration $h + 1$. This single change leads to many different possible branches in the protocol's execution when processing a *propose* message (addressed during the presentation of the protocol's specification in Section 5.3) and demonstrate the complexity of achieving liveness guarantees in a probabilistic SMR protocol. Despite this, this change is not sufficient to ensure ProSimplex's liveness. Consider the following situation, where we assume a correct process i

the leader of iteration h and a process j the leader of iteration $h + 1$. Additionally, consider that process j crashes in the beginning of iteration h, or is Byzantine and remains silent for the remaining of the protocol's execution at that point. If during iteration h, process i proposes a new block b and a subset S of fewer than $2f + 1$ processes fail to observe a probabilistic quorum of *vote* messages for b, all processes in S will not be able advance to iteration $h + 1$ either by forming a deterministic quorum of *timeout* messages for $h + 1$ or by receiving the proposal for iteration $h + 1$ from process j. To circumvent this issue, ProSimplex allows a correct process that advanced to iteration $h + 1$, to send a *reply* message containing the block of iteration h and its corresponding probabilistic quorum certificate to some process of S, p, when it observes a *timeout* message for iteration $h + 1$ from p. This mechanism is not necessary in the Practical Simplex protocol because in Practical Simplex processes broadcast *vote* messages to all other processes instead of a selected sample of recipients.

5.3 Protocol Specification

We now present a full description of the ProSimplex's protocol. The pseudocode of ProSimplex is presented in Algorithm 1. For simplicity, when referring to some block's iteration $b.header.h$, we use $b.h$ instead.

ProSimplex proceeds in a sequence of iterations. At the beginning of the protocol, all correct processes initialize a variable $iter$ with value 1 (line 1) and enter the first iteration (line 2). Upon entering a new iteration h, a correct process stores h in $iter$ and resets the countdown of its local timer by calling the `resetTimer` function and by setting a flag $isTimeout$ to $false$ (lines 8-10). If the timer of a correct process i expires during an iteration h, process i stops the timer's countdown by calling the `stopTimer` function, sets the $isTimeout$ flag to true and sends a *timeout* message for iteration $h + 1$ to all other processes (lines 14-16). During each iteration, a pre-determined process is assigned the role of leader and is responsible for proposing a new block to be added to the blockchain. All processes in the system can determine the leader of an iteration h with the `leader` predicate.

$$\texttt{leader}(h) = H(h) \bmod n$$

In the propose phase, the leader of iteration h, L_h, gathers a set of unconfirmed transactions, txs, and creates a new block proposal b for the current iteration, $iter$, extending the last notarized block (highest iteration and length) of its local notarized blockchain, $b_{|\mathcal{B}|}$ (line 12). Afterwards, the leader collects a probabilistic quorum of vote signatures on the last notarized block and compiles them into a notarization certificate and broadcasts a *propose* message, $\langle \text{PROPOSE}, b, b_{|\mathcal{B}|}.cert, b_{|\mathcal{B}|}.h \rangle_{L_h}$, containing the newly generated block b and the iteration and quorum certificate of the last notarized block (line 13). Upon receiving a *propose* message for a block b from a process j for iteration $b.h$, a correct process verifies the validity of the proposal by checking if j is the leader of iteration $b.h$ and if it is the first proposal received for iteration $b.h$ (line 17). Once a proposal has been validated, there are three different possible scenarios. In the first and most common scenario, a correct process votes for the proposal of iteration $b.h$ provided that the proposed block's iteration matches its current iteration, the timer for the current iteration has not yet expired, and the proposed block b satisfies the `extendable` predicate (line 27).

$$\texttt{extendable}(b) \iff b_{|\mathcal{B}|}.k = b.k - 1 \wedge b.hash = H(b_{|\mathcal{B}|}.header) \wedge b_{|\mathcal{B}|}.h < b.h$$

Algorithm 1 ProSimplex – process i.

task `start`()
1: $iter \leftarrow 1$
2: **trigger** `newIteration`($iter$)
task `handleNotarization`($header, cert, txs, h$)
3: `notarize`($header, cert, txs$)
4: **if** $\neg isTimeout$
5: $S_f, P_f \leftarrow$ `VRF_prove`($K_{p,i}, h \mid\mid$ "finalize", $o \times q$, `leader`$(h+1)$)
6: $\forall p \in S_f$, **send** $\langle$FINALIZE, $h, S_f, P_f\rangle_i$ **to** p
7: **trigger** `newIteration`($h+1$)
upon `newIteration`(h)
8: $iter \leftarrow h$
9: `resetTimer`($timer$)
10: $isTimeout \leftarrow false$
11: **if** $i =$ `leader`(h)
12: $b \leftarrow \langle\langle H(b_{|\mathcal{B}|}.header), b_{|\mathcal{B}|}.k+1, iter, H(txs)\rangle, txs\rangle$
13: **broadcast** $\langle$PROPOSE, $b, b_{|\mathcal{B}|}.cert, b_{|\mathcal{B}|}.h\rangle_i$
upon expiring $timer$
14: `stopTimer`($timer$)
15: $isTimeout \leftarrow true$
16: **broadcast** $\langle$TIMEOUT, $iter+1\rangle_i$
upon receiving $\langle$PROPOSE, $b, cert, h\rangle_j$
17: **if** $proposes[b.h] = \perp \wedge j =$ `leader`$(b.h)$
18: $proposes[b.h] \leftarrow b$
19: **if** $b.h > iter$
20: **if** $proposes[h] \neq \perp \wedge$ `validCert`$(cert, proposes[h].header)$
21: **if** $iter = h$
22: `handleNotarization`($proposes$[h].header, $cert, proposes[it].txs$, h)
23: **else**
24: **send** $\langle$REQUEST, $|\mathcal{B}|\rangle_i$ **to** j
25: **else if** $proposes[h] = \perp$
26: **send** $\langle$REQUEST, $|\mathcal{B}|\rangle_i$ **to** j
27: **pre:** $b.h = iter \wedge$ `extendable`$(b) \wedge \neg$*isTimeout*
28: $S_v, P_v \leftarrow$ `VRF_prove`($K_{p,i}, iter \mid\mid$ "vote", $o \times q$, `leader`$(iter+1)$)
29: $\forall p \in S_v$, **send** $\langle$VOTE, $\langle b.header\rangle_i, S_v, P_v\rangle$ **to** p
upon receiving $\{\langle$VOTE, $\langle header\rangle_j, S, P\rangle : j \in Q\} = V$ **from** a probabilistic quorum Q
30: **pre:** $header.h = iter \wedge proposes[iter] \neq \perp \wedge$ `extendable`$(proposes[iter]) \wedge (\forall\langle \_, \_, S, P\rangle_j \in V : i \in S \wedge$ `VRF_verify`$(K_{u,j}, iter \mid\mid$"vote", $o \times q$, `leader`$(iter+1), S, P))$
31: `handleNotarization`($header, \{\langle header\rangle_j \mid \langle \_, \langle header\rangle_j, \_, \_\rangle \in V\}, proposes[iter].txs, iter$)
upon receiving $\{\langle$FINALIZE, $h, S, P\rangle_j : j \in Q\} = F$ **from** a probabilistic quorum Q
32: **pre:** $\exists b \in \mathcal{B}, b.h = h \wedge (\forall\langle \_, \_, S, P\rangle_j \in F : i \in S \wedge$ `VRF_verify`$(K_{u,j}, iter \mid\mid$ "finalize", $o \times q$, `leader`$(h+1), S, P))$
33: `finalize`(h)
upon receiving $\langle$TIMEOUT, $nextIter\rangle_j$ **from** *j* for the first time
34: **if** $\exists b \in \mathcal{B}, b.h = nextIter - 1$
35: **send** $\langle$REPLY, $\{\langle b.header, b.cert, b.txs\rangle\}\rangle_i$ **to** j
upon receiving $\{\langle$TIMEOUT, $nextIter\rangle_j : j \in Q\}$ **from** a quorum Q
36: **pre:** $nextIter = iter + 1$
37: **trigger** `newIteration`($nextIter$)

Algorithm 1 (continued) ProSimplex – process i.

upon receiving $\langle \text{REQUEST}, l \rangle_j$
38: **if** $l < |\mathcal{B}|$
39: **send** $\langle \text{REPLY}, \{\langle b_k.header, b_k.cert, b_k.txs \rangle\}_{k=l}^{|\mathcal{B}|} \rangle_i$ **to** j
upon receiving $\langle \text{REPLY}, \mathcal{M} = \{\langle b_k.header, b_k.cert, b_k.txs \rangle\}_{k=|\mathcal{B}|}^{l} \rangle_j$
40: **if** $l \geq |\mathcal{B}| \wedge \forall \langle header, cert, txs \rangle \in \mathcal{M}, (\texttt{validCert}(cert, header) \wedge header.txs = H(txs))$
41: **for** each $\langle header, cert, txs \rangle \in \mathcal{M}$
42: **if** $\nexists b \in \mathcal{B}, b.h = header.h$
43: **if** $proposes[b.h] = \perp$
44: $proposes[b.h] \leftarrow \langle header, txs \rangle$
45: **if** $iter = header.h$
46: $\texttt{handleNotarization}(header, cert, txs, header.h)$
47: **else**
48: $\texttt{notarize}(header, cert, txs)$

Afterwards, a correct process uses the VRF to select a random sample S_v to which it sends a *vote* message containing a digital signature over the proposed block's header (lines 28-29). The second and third cases correspond to the scenarios in which a process has still not reached iteration $b.h$ (line 19). If a correct process's current iteration matches the last notarized block's iteration, h, included in the *propose* message, and has received a proposal for iteration h, it verifies the quorum certificate of the last notarized block, $cert$, included in the *propose* message using the `validCert` predicate (lines 20-21). In the case where the certificate is valid, the process concludes that it has not received a probabilistic quorum of *vote* messages for the last notarized block of iteration h, and advances to the next iteration by notarizing the block proposal of iteration h (line 22). At this point, the process may also vote for the proposal of iteration $b.h$ if the conditions for voting are met. On the other hand, if either the process has no stored proposal for iteration h (lines 25-26), or it has one and $cert$ satisfies the `validCert` predicate while its current iteration differs from h (lines 23-24), then the process is lagging behind in the protocol's execution and it sends a *request* message to j containing its local notarized blockchain's length, $|\mathcal{B}|$.

$$\texttt{validCert}(cert, header) \iff |\{j \mid \langle \_, \_, \langle h \rangle_j \rangle \in cert, h = header\}| \geq n - f$$

All correct processes wait until receiving a set of *vote* messages containing a signature over a proposed block's header from a probabilistic quorum or a set of *timeout* messages from a deterministic quorum, during each iteration. Upon receiving a probabilistic quorum of *vote* messages for a proposed block b of an iteration h, a correct process i will eventually notarize b when its current iteration matches h and it has received a *propose* message for its current iteration. Process i only notarizes b if it satisfies the `extendable` predicate and if all votes for b in the probabilistic quorum contain a valid sample S generated using the VRF, that includes i (lines 30-31). When a notarization for a block b occurs for iteration h, a correct process adds b to its notarized blockchain by calling the `notarize` function, uses the VRF to select a random sample S_f to which it sends a *finalize* message for iteration h if the timer for iteration h has not yet expired, and starts iteration $h + 1$ (lines 3-7). Alternatively, if a correct process observes a deterministic quorum of *timeout* messages for iteration $h + 1$ before notarizing a block for iteration h, it simply starts iteration $h + 1$ (lines 33-37). When a process receives a *timeout* message for iteration $h + 1$ from a process j for the first time, it sends a *reply* message to j containing the block of iteration h and its corresponding probabilistic quorum certificate if it is present in its notarized blockchain (lines 34-33).

$$\texttt{notarize}(header, cert, txs) : \mathcal{B} \leftarrow \mathcal{B} \cup \langle header, cert, txs \rangle$$

At any given point during the protocol's execution, if a correct process receives a probabilistic quorum of *finalize* messages for iteration h, it will eventually finalize the block of iteration h as well as its notarized parent chain. A correct process i only finalizes iteration h by calling the `finalize` function, if it contains a block for iteration h in its current blockchain and all *finalize* messages for h in the probabilistic quorum contain a valid sample S generated using the VRF, that includes i (lines 32-33). After a block b becomes finalized for some process i, all other processes will eventually finalize b, becoming final and immutable in the blockchain. Furthermore, every subsequent block proposal is guaranteed to extend b's chain.

$$\texttt{finalize}(h) : \text{mark block } b \text{ satisfying } b.h = h \text{ and its parent chain finalized}$$

The algorithm also incorporates a state recovery mechanism to handle recoveries or delays. Each time a correct process receives a *request* message from a process j, containing the current length of j's blockchain, l, it replies by sending a *reply* message to j containing all blocks present in its notarized blockchain with length equal or higher than l, along with their respective probabilistic quorum certificates (lines 36-37). Accordingly, when a correct process receives a *reply* message containing only blocks with length equal or higher to the current length of its notarized blockchain, it verifies if each of those blocks' probabilistic quorum certificate satisfy the `validCert` predicate and if the hash of the transactions present in the block's header match the received transactions (line 38). If all of these conditions are met, a correct process notarizes all blocks in the *reply* message for which it has not yet observed a notarization in their corresponding iteration. Moreover, if the iteration of a notarized block matches the process's current iteration, the process also advances to the next iteration as usual (lines 39-48).

5.4 Message and Communication Complexity

ProSimplex's message complexity per block is $O(n\sqrt{n})$ in the best-case-scenario: $O(n)$ for *propose* messages, $O(n\sqrt{n})$ for *vote* messages, and $O(n\sqrt{n})$ for *finalize* messages. In every iteration except the first, the leader broadcasts a *propose* message containing a probabilistic quorum certificate to all processes. Thus, the communication complexity of the propose phase is $O(n\sqrt{n})$, and the overall communication complexity per block of ProSimplex remains $O(n\sqrt{n})$, an improvement over ProBFT's communication complexity of $O(n^2\sqrt{n})$.

5.5 Outline of Correctness Proofs

ProBFT and ProSimplex are similar in many aspects, but differ in a few. In both protocols, in each iteration h, (1) the leader of iteration h sends its proposal block to all processes, (2) upon receiving a valid block from the leader of h, each process selects a random sample of processes and sends its *vote* message to them, (3) upon receiving *vote* messages from a probabilistic quorum of processes, each process notarizes the proposed block and sends a *finalize* message to a random sample of processes, and (4) upon receiving *finalize* messages from a probabilistic quorum of processes, each process finalizes the

proposed block. This similarity enables us to extend the safety and liveness guarantees of ProBFT to ProSimplex. Besides, in both protocols, if a leader is faulty or slow, processes employ a deterministic mechanism to change the iteration.

The key distinction between these protocols lies in their consensus models: ProBFT is a single-shot consensus protocol, i.e., correct processes only decide a single value. In contrast, ProSimplex is a multi-shot protocol: once the leader of iteration $h + 1$ notarizes the block proposed in iteration h, it can immediately propose a new block. Recall that to ensure the leader of iteration $h + 1$ can notarize the block proposed in iteration h, each correct process, in addition to sending its *vote* message to a random sample of processes, also sends it to the leader of iteration $h + 1$.

Liveness analysis. We need to show that if a valid input *txs* is provided to every correct process, then it eventually will be added to the finalized chains of all correct processes with high probability. With this aim, suppose iteration h is the first iteration after GST whose leader is a correct process. There are two possible cases:

Case–1. The leader of h has notarized block b proposed in iteration $h - 1$. In this case, the leader proposes a new block b' extending b. If correct processes do not have b (or preceding blocks), by executing the recovery sub-protocol, they can recover their missing notarized blocks; then, they can vote for b'. Consequently, in this case, all correct processes notarize b' and send their *finalize* messages with high probability; as a result, all correct processes finalize b' with high probability.

Case–2. The leader of h enters iteration h with *timeout* messages. There are two possible sub-cases:

Sub-case–2.1. The leader of h has not notarized block b with the greatest height; however, there is at least a correct process i that has notarized b. Note that i sends b to the leader of h through a *reply* message. Consequently, the leader of h can recover its missing notarized blocks and propose a new block extending b. The remaining part of this sub-case is identical to the first case.

Sub-case–2.2. There is no correct process that has notarized block b with the greatest height, and only Byzantine processes have notarized b. Some Byzantine process might send b to the leader of h through a *reply* message; however, when the leader of h sends a *request* message to recover b, the Byzantine processes might remain silent. If the leader cannot recover b until the expiration of a *recovery timeout*, it can safely conclude that the process holding b is Byzantine, and there is no need to wait for b. The leader can then remove b from the list of received blocks obtained through *reply* messages and select the block with the greatest height from the updated list. Again, one of the sub-cases can occur. This situation continues until the leader can extend a block.

From the above cases, it suffices to consider Case–1, which is exactly the value computed by ProBFT (Theorem 3).

Theorem 3. After GST, if the leader of iteration h is correct, then each correct process finalizes a value in iteration $h + 1$ with a probability of at least $1 - 2(n - f) \cdot \exp(-\Theta(\sqrt{n}))$.

Safety analysis. In ProBFT, a correct process finalizes (commits in ProBFT parlance) a block if it first notarizes (prepares in ProBFT parlance) the block (i.e., receives *vote* messages from a probabilistic quorum) and then finalizes it (i.e., receives *finalize* messages from a probabilistic quorum). It is essential to note that both of these steps are randomized. In ProSimplex, however, the first step may not be randomized, but the second step is still randomized and similar to that in ProBFT. In what follows, we elaborate on the first step to explain why it may not be randomized.

Recall that in each iteration h, each correct process, in addition to sending its *vote* message to a random sample of processes, also sends it to the leader of iteration $h + 1$. Suppose the leaders of iterations h and $h + 1$ are Byzantine. Further, assume the leader of iteration h proposes an invalid block or multiple blocks and sends its proposal only to Byzantine processes. Then, Byzantine processes send their *vote* messages to the leader of iteration $h + 1$. Since the leader of iteration $h + 1$ needs to receive *vote* messages from a probabilistic quorum of processes to notarize the block, Byzantine processes can notarize an invalid block or even multiple blocks without any randomization. Compare this with ProBFT, where each process, including any Byzantine process, must belong to a sufficient number of random samples in order to receive *vote* messages from a probabilistic quorum of processes (i.e., receiving *vote* messages from a probabilistic quorum of processes is a randomized step).

This difference between ProBFT and ProSimplex reduces the per-iteration probability of ensuring safety in ProSimplex compared to ProBFT. However, ProSimplex is a chained protocol, i.e., each notarized block extends a previously notarized block. Accordingly, even in the worst-case scenario, any two consecutive finalized blocks in ProSimplex can be regarded as a single finalized block in ProBFT. Hence, the same safety probability of ProBFT (Corollary 1) applies to the first finalized block in any sequence of two consecutive finalized blocks.

Theorem 4. For any two consecutive finalized blocks, the probability that the first block causes a safety violation is at most $\exp(-\Theta(\sqrt{n}))$.

Chapter 6

Implementation

The previous chapters presented the formal specifications of the Practical Simplex and ProSimplex protocols (Chapter 4 and Chapter 5, respectively). In this chapter, we turn to the practical side of these protocols by discussing the technical details of their developed implementations. Our goal is to describe how both protocols can be implemented in a concrete system. We implement these two prototypes to experimentally evaluate the throughput and latency of Practical Simplex and ProSimplex.

The Practical Simplex and ProSimplex protocols are implemented using the Rust programming language and the source code is available on a Github repository.[1] Rust was chosen as the implementation's programming language because it combines high performance and strong memory safety guarantees enforced at compile time, thereby reducing the risk of possible bugs, particularly, concurrency problems and memory leaks likely to arise in the implementation of a BFT consensus protocol that is intended to run indefinitely.

This section is organized as follows. First, an overview of the overall architecture of the implementation and the software tools employed is presented. Then, the main data structures defined to implement both protocols are described. Additionally, this section details how the previously specified VRF was implemented, followed by a discussion of the strategies used to achieve garbage collection necessary for long executions of the protocols. Finally, the challenges encountered during the implementation of both protocols are discussed.

6.1 Initialization

The initialization phase corresponds to the setup that processes must complete before executing the protocols. The implementations of the Practical Simplex and ProSimplex protocols allow a system composed of $n = 3f + 1$ processes, in which up to f processes can be faulty. At the beginning of execution, each process knows the IDs, hostnames, and ports of all other processes in the system. This information can be configured through a configuration file. In addition, the ProSimplex implementation defines the constants used for configuring the size of the probabilistic quorums and recipient samples. Private keys are provided as system variables, while public keys are publicly available in a configuration file. The size of transactions and the number of transactions per block must be specified when executing both protocols. The implementations of both protocols rely on Ed25519 key pairs, supported by the `ed25519_dalek` [32] crate, which is also employed in other efficient implementations of BFT consensus protocols (e.g., [31]). This library was selected because it enables fast verification of individual

[1] Source code available at: https://github.com/AndreSantos0/Probabilistic-Chained-Blockchain-Consensus

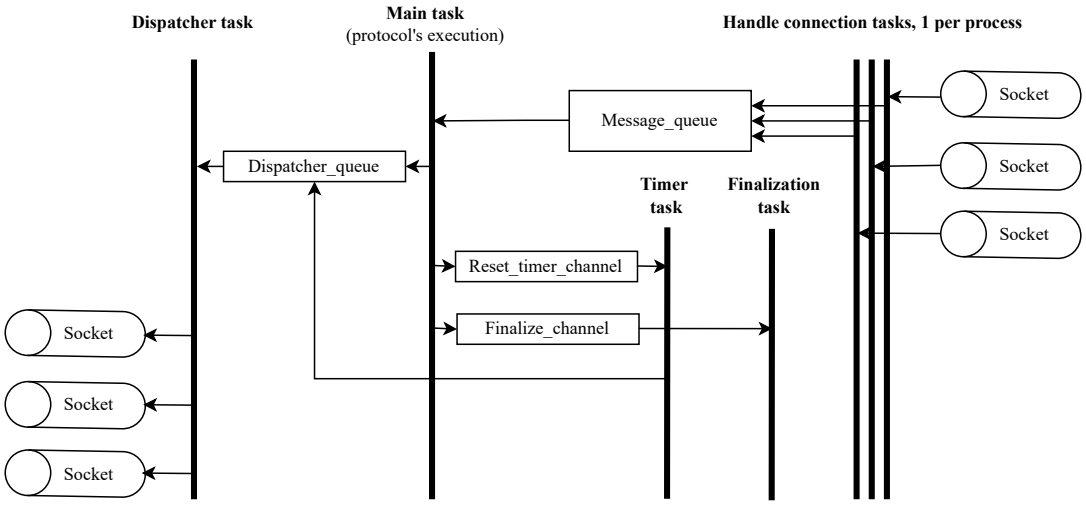

Figure 6.1: ProSimplex and Practical Simplex's implementation architecture.

message's signatures as well as batch verification of multiple signatures, useful when validating block certificates. During the initialization phase, each process loads the public keys of all processes in the system, as well as its own private key, which is used to sign outgoing messages. Before starting the consensus protocol, each process broadcasts a signed nonce to all other processes. This step allows every process to associate process IDs with their corresponding public keys by verifying the received nonces with the preloaded public keys. Once a process has received and successfully verified signed nonces from $n - 1$ distinct processes, it begins executing the consensus protocol.

6.2 Architectural Overview

In both the ProSimplex and Practical Simplex protocols, each process must perform several tasks concurrently: sending messages to other processes, maintaining and decrementing a local timer for each iteration, processing incoming messages, and outputting finalized blocks. These tasks are interdependent and must be handled without blocking one another, so that the protocol's throughput is not reduced and its latency is not increased. To achieve this, the implementation makes use of Tokio [11], an asynchronous multi-threaded runtime for Rust, which enables lightweight task parallelization and safe communication between concurrent tasks through channels. In Tokio, a task is a lightweight non-blocking unit of execution that represents an asynchronous computation. Tasks are scheduled by the Tokio runtime and may be executed on different threads. The Tokio runtime is well-suited for implementing BFT consensus protocols because it provides high performance and scalability, both of which are essential when handling a large number of processes that exchange many messages during each protocol's phase.

Figure 6.1 represents the implementation's architectural design and the interaction between the multiple concurrent tasks. Tasks communicate and synchronize their actions with each other using communication channels. As shown, the focus lies on the protocol's execution rather than client interaction. Consequently, transactions are not submitted by clients but are instead generated at the time of a new proposal. Cryptographic operations are performed concurrently with the protocol's execution to enable faster message processing and reduce the overall finalization latency of the implementation.

The communication between processes is established through TCP sockets and processes communicate with each other using the messages defined at the application's level. Application-level messages are serialized into bytes before being written to a socket, and incoming bytes are deserialized back into application-level messages upon reception. The connections between processes are maintained open throughout the entirety of the protocol's execution. Each process creates a dedicated task to handle incoming messages for each other process in the system. Upon receiving data, the message is first deserialized. If the deserialization operation succeeds and the resulting message conforms to the application's specification, several verification steps are performed: the message's signature is verified using the public key of the sending process, block certificates and transactions contained in *reply* messages are validated, and exclusively in the ProSimplex protocol, when receiving *vote* and *finalize* messages, processes use the VRF to verify if they are indeed part of the recipient sample using the proof received. A message is added to the message queue only after all these checks succeed.

The messages present in the message queue are consumed by the main task. The main task is responsible for the execution of the protocols' main logic. The implementation of the protocol's main logic follows an event-based architecture and events are triggered according to the messages polled from the message queue. Since the protocol operates in an asynchronous environment, the order of incoming messages cannot be predicted, making message storage necessary. The main task maintains all data structures that hold the content of received messages. Specifically, these structures track proposed blocks and their associated transactions for each iteration, the votes received for specific blocks, the IDs of processes that have sent a *timeout* or *finalize* message in a given iteration, and exclusively in the ProSimplex protocol, the probabilistic quorum certificates for the last notarized block included in *propose* messages. Together, these structures ensure that all protocol messages can be retrieved whenever required for advancing the protocol's execution. Moreover, these structures are not shared across any of the concurrent tasks to avoid unnecessary additional complexity regarding concurrency control. The main task signals the timer task each time an iteration advances during the protocol's execution by sending a message through the reset-timer channel. When the timer task receives a message from this channel, it resets the iteration timer and starts a new countdown.

Each time a block is finalized during the protocol's execution, the main task sends the content of the finalized block, along with its corresponding vote certificate, through the finalize channel. The finalization task then consumes the finalized blocks from this channel and writes them to a JavaScript Object Notation (JSON) file, representing the finalized blockchain.

The dispatcher task is responsible for signing and sending messages to other processes. The main task adds a message to the dispatcher queue each time the protocol's specification requires it, while the timeout task similarly adds a *timeout* message whenever a timeout occurs. In the ProSimplex protocol, for *vote* and *finalize* messages, the dispatcher task uses the VRF to generate a recipient sample containing the IDs of the processes to which the message must be sent, along with a proof that the sample was generated using the VRF.

6.3 Data Structures

Figure 6.2 depicts a UML diagram defining the data structures used to implement the Practical Simplex and ProSimplex protocols. The implementation follows an object-oriented design in the sense that, although Rust does not refer to structs and enumerations as objects, both can encapsulate data and provide methods.

In the implementation, a process is represented by the *Node* data structure, which stores its host

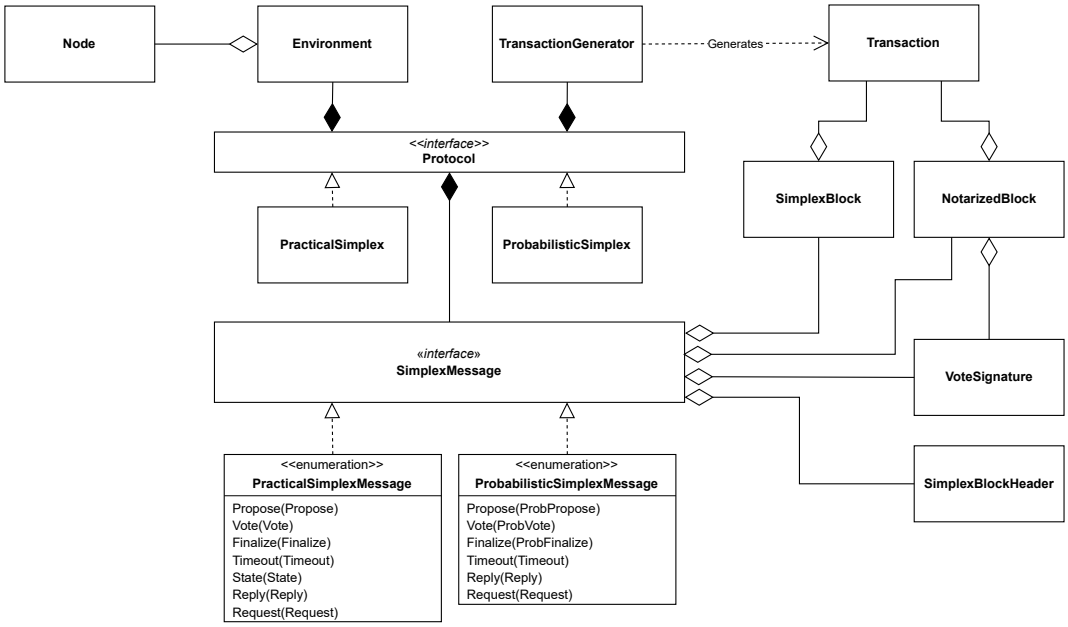

Figure 6.2: UML diagram of Practical Simplex and ProSimplex's implementation.

address, port, and respective ID in the distributed system. The *Environment* structure maintains the configuration of the whole system, including the information of all processes, the size of transactions, and the number of transactions per block. Each protocol execution instantiates a *TransactionGenerator*, which simulates the submission of transactions by clients. Finally, a *VoteSignature* binds a signature contained in a *vote* message to the ID of the process that cast the vote, and a set of *VoteSignature* is used to represent either deterministic or probabilistic block certificates.

As previously mentioned, *PracticalSimplex* and *ProbabilisticSimplex* utilize different mappings to maintain all messages received. Each of these mappings is a HashMap data structure and they associate proposed *SimplexBlock*'s and their corresponding transactions with their respective iteration, the received *VoteSignature*'s for each *SimplexBlockHeader* and, the IDs of processes that have sent a *timeout* or *finalize* message with the corresponding iteration included in the message. Consequently, a specific Blockchain data structure containing notarized blocks is not necessary since all notarized blocks can be easily identified by filtering all *SimplexBlockHeader* that received a deterministic or probabilistic quorum of valid *VoteSignature*'s, avoiding data duplication in memory. Additionally, when finalizing a chain of notarized blocks, a parent block can be found by matching the parent hash of a block with the hash of blocks from earlier iterations, present in the mappings.

6.4 Verifiable Random Function

The goal of implementing VRF is to use it to deterministically generate a random sample of process IDs along with providing a cryptographic proof that the sample was generated using the VRF. As mentioned previously, the VRF has two functions, `VRF_prove` and `VRF_verify`. The `VRF_prove` function takes a process's private key, a seed string, the total number of processes n, the sample size s, and the leader's ID. It first excludes the leader from the list of possible candidate processes, and signs the seed using the process's private key to produce a proof. This proof is hashed using the SHA-256 hash function

to generate a seed used to instantiate a deterministic random number generator. The random number generator is used to shuffle the candidate IDs. The function then selects $s - 1$ IDs from the shuffled list and adds the leader's ID to form the final sample set, returning both the sample and the proof. The `VRF_verify` function works in mirrored manner. Firstly, it uses the sender's public key to check that the proof correctly signs the seed. Then it replicates the generation of the sample to compute a expected sample set. If the verified sample matches the expected one, the function returns true, otherwise, it returns false.

6.5 Garbage Collection

Since the protocols run indefinitely, it is necessary to clear accumulating data originated from the storage of messages observed during each iteration to avoid the crashing of processes. In the Practical Simplex protocol, *propose* and *finalize* messages can be removed from memory once their respective iterations are finalized without affecting the correctness of the implementation. In ProSimplex, however, this is not possible for *propose* messages. When a process does not observe a probabilistic quorum of *vote* messages for the last notarized block b, it may need to verify the signatures of b's certificate included in the following proposal, which requires receiving and storing the proposal of b. If proposals were always removed from memory once their respective iterations are finalized, a process would need to repeatedly request the block and its transactions from other processes when it does not observe a probabilistic quorum of *vote* messages for the last notarized block (Algorithm 1, lines 25-26). To avoid this overhead, the ProSimplex implementation only removes proposals from memory once their iteration is sufficiently older than the iteration currently being finalized, ensuring that processes rarely, if ever, need to send a request. Additionally, in ProSimplex, the certificates of the last notarized blocks included in *propose* messages are also removed from memory once the protocol advances to their corresponding iterations. In the same way, in both protocols, *timeout* messages can only be discarded once the protocol has reached their iteration number. Finally, if a finalization for an iteration h occurs, all vote signatures and transactions for notarized blocks proposed in iteration h or earlier are removed from memory, with the exception of the vote signatures for the block of iteration h. The last notarized block of iteration h and its corresponding quorum of vote signatures, despite being finalized, must be kept in memory to ensure the validity of future proposals, since when a new block is proposed or extended, its parent hash is compared against the hash of the last notarized block.

6.6 Challenges

Although both Practical Simplex and ProSimplex were designed to be simple and build upon a straightforward protocol (Simplex [9]), their implementation proved to be a difficult and time-consuming process. A substantial amount of time was spent handling the wide variety of corner cases that arise in asynchronous distributed systems, particularly due to the lack of message ordering. This often required rethinking and refactoring different parts of the protocols' specifications and their implementations in order to preserve safety and liveness properties. In addition to ensuring correctness, considerable effort was dedicated to optimizing their implementations. This required refactoring the architecture to improve performance by parallelizing the multiple interdependent tasks, and analyze the code base regularly to eliminate unnecessary complexity. Additionally, it was necessary to design the implementations in a sufficiently generic manner to eliminate redundancy arising from the similarities between the two protocols. These challenges were further amplified by the fact that this was a first-time experience with the

Rust programming language. Overall, this work highlights that even protocols designed to be simple can present considerable implementation challenges.

Chapter 7

Evaluation

In this chapter, we evaluate the performance of the implementations of Practical Simplex and ProSimplex, described in Chapter 6. The key metrics evaluated are throughput and finalization latency. With this evaluation, we aim to analyze and compare the performance of the two protocols in systems with different numbers of processes, to identify in which scenarios using ProSimplex is justified, by evaluating how much better it scales compared to Practical Simplex. Additionally, we compare both protocols with Jolteon [15], however, instead of using the official Jolteon implementation [31], we integrate Jolteon into the same code-base used for Practical Simplex and ProSimplex. This ensures a fair comparison by eliminating differences caused by implementation quality or optimizations, allowing the evaluation to reflect the relative performance of the protocols themselves. We compare both protocols with Jolteon because it is an improved version of HotStuff [35] and we are interested in comparing these protocols with another protocol that achieves linear message complexity.

Initially, this chapter describes the experimental setup utilized during the evaluation process. Then, it presents the results obtained for the three protocols across different system and transaction sizes, as well as varying numbers of transactions per block. Finally, it presents and discusses the results of the experimental evaluation.

7.1 Experimental Setup

The experiments were performed on a cluster consisting of Dell PowerEdge R410 machines, each equipped with two Intel Xeon E5520 CPUs (quad-core, 2 threads per core, 16 hardware threads in total) running at 2.27 GHz, and 32 GB of RAM. The machines were connected through a gigabit Ethernet network. The operating system used by all machines during the evaluation was Ubuntu 20.04.

Since the cluster utilized for the experiments consists of only 10 physical machines, for experiments requiring more than 10 processes we simulate multiple processes on each machine, distributed equally. For example, in the largest configuration with $n = 100$, each physical machine executes 10 different processes. This configuration is necessary to evaluate scalability for larger system sizes despite the limited number of physical machines available. Nonetheless, the local resources of a single server (CPU and memory) were not exhausted when running multiple processes in the same machine.

7.2 Performance Analysis

The following experiments were conducted in scenarios with blocks containing either 1000 or 10000 transactions with a size of 242 B or 992 B, and with different system sizes of 4, 7, 10, 34, 67, and 100

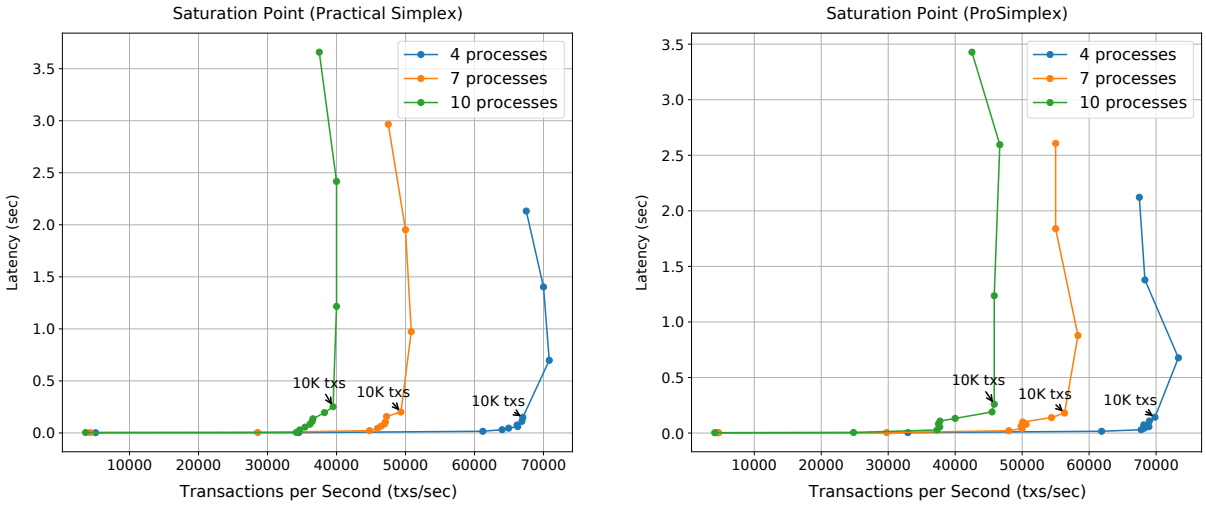

Figure 7.1: Saturation point for the number of transactions per block for Practical Simplex and ProSimplex.

processes. In these experiments, Byzantine behavior is not simulated and we consider only the the most optimal scenario of all protocols. For ProSimplex, the presented results were obtained for $o = 1.7$ and $l = 2.0$, which determine the size of recipient samples and probabilistic quorums presented in Table 7.1.

In general, throughput tends to increase as the number of transactions per block grows. For this reason, we specifically select blocks containing 10000 transactions for our experiments (which we justify below) and compare it with a smaller payload of 1000 transactions. Prior to conducting the experiments, we determine the point at which further increasing the number of transactions per block no longer results in meaningful throughput improvements due to excessive latency spikes. We call this point the saturation point. In Figure 7.1, we determine the saturation point for the number of transactions per block the Practical Simplex and ProSimplex protocols by plotting finalization latency in function of transactions-per-second. The transactions-per-second values correspond to different block sizes of 10, 100, 1000, 2000, 3000, 4000, 5000, 7500, 10000, 50000, 100000, and 150000 transactions of 242 B. From this analysis, we conclude that for both protocols, latency spikes occur at scenarios considering block sizes bigger than 10000 transactions.

Table 7.2 calculates the total bandwidth usage by all three protocols during the propose phase, for all the different scenarios considered during the experiments. The values indicate the total number of bytes sent by the leader process during the propose phase. The highlighted values denote cases where the bandwidth exceeds the limit of the gigabit network utilized in this experimental setup. Specifically, this occurs for blocks containing 10000 transactions of size 992 B in systems with 34, 67, and 100 processes. We present this table to provide additional context for interpreting the following experimental results for

Table 7.1: Probabilistic quorum and recipient sample sizes for $\ell = 2$ and $o = 1.7$.

System Size n	**Probabilistic Quorum Size** $\lfloor q \rfloor$	**Recipient Sample Size** $\lfloor \min(o \times q, n) \rfloor$
4	4	4
7	5	7
10	6	10
34	11	19
67	16	27
100	20	34

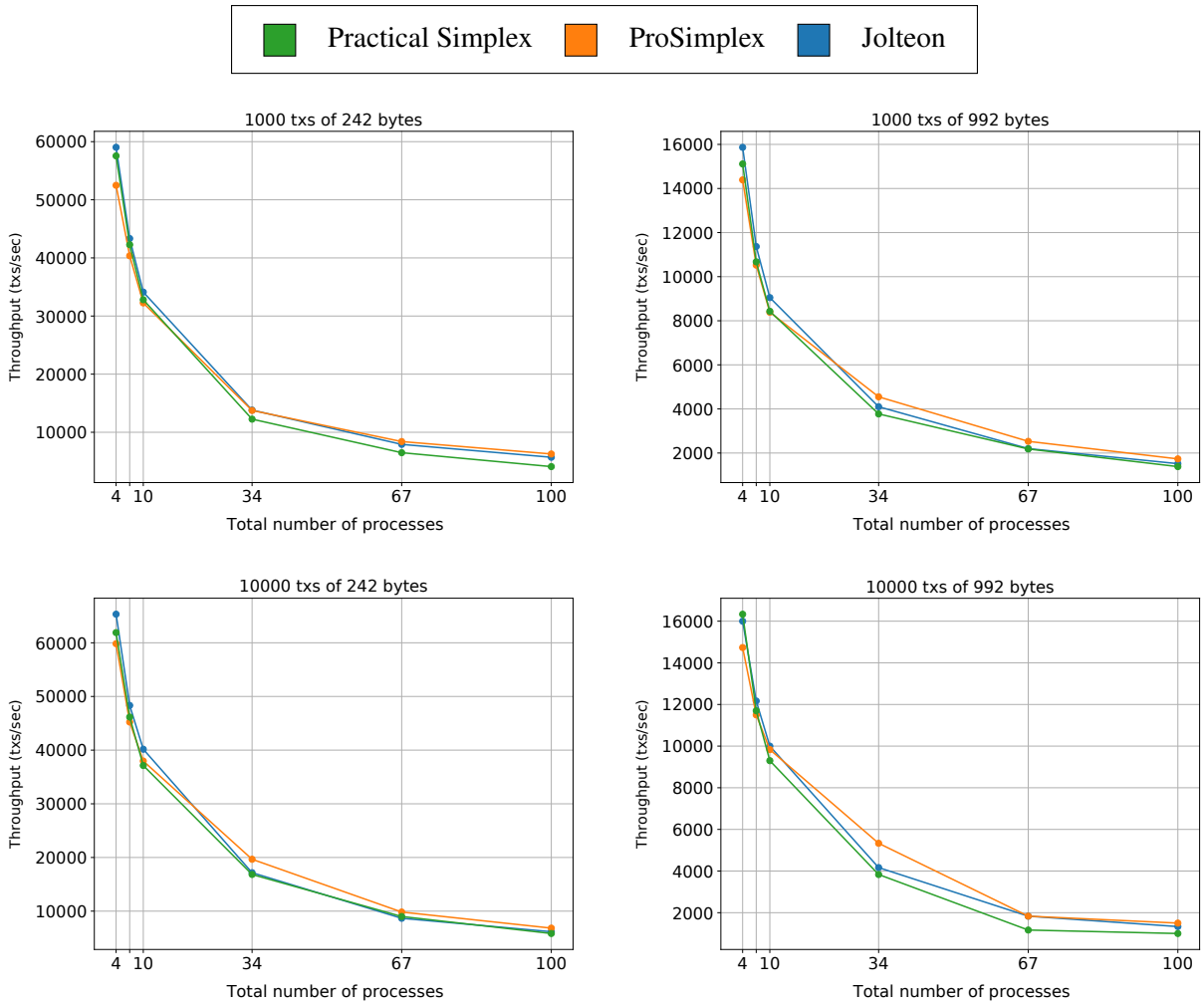

Figure 7.2: Throughputs of Practical Simplex, ProSimplex and Jolteon with increasing transaction payload size for blocks with 1000 and 10000 transactions.

throughput and finalization latency.

Figure 7.2 presents the average throughput results of Practical Simplex, ProSimplex, and Jolteon for the multiple executions of each protocol for the different experimental scenarios. Standard deviation is not presented because the experimental results were highly consistent across repetitions. Throughput is calculated by dividing the number of finalized/committed transactions by the execution time of the experiment. Figure 7.3 shows the finalization latency of the three protocols. We define finalization latency as the time interval between the start of an iteration (or view) and the moment when a block becomes finalized (or committed) for that iteration (or view). The depicted finalization latency corresponds to the average of all such intervals stored during the protocols' execution. The presented throughput and

Table 7.2: Total bandwidth usage during the propose phase (highlighted values exceed the limit of the gigabit network).

#Txs/Block	Txs Size	4	7	10	34	67	100
1000	242 B	0.97 MB	1.69 MB	2.42 MB	8.23 MB	16.21 MB	24.20 MB
1000	992 B	3.97 MB	6.94 MB	9.92 MB	33.73 MB	66.44 MB	99.20 MB
10000	242 B	9.68 MB	16.94 MB	24.20 MB	82.28 MB	162.14 MB	242.00 MB
10000	992 B	39.68 MB	69.44 MB	99.20 MB	**0.34 GB**	**0.66 GB**	**0.99 GB**

finalization latency graphs are the result of ten executions for each configuration.

The throughput results (also detailed in Appendix B, Table B.1) exhibit clear trends that can be generalized across all scenarios. As the number of processes increases, the throughput consistently decreases due the additional number of messages each protocol must exchange and process during the protocol's execution. The number of transactions-per-second also becomes lower as the size of transactions grows. Larger transactions increase the size of each block and, consequently, the amount of data that needs to be transmitted per proposal, amplifying the total bandwidth usage.

The graphs of Figure 7.2 show that ProSimplex scales better in terms of throughput than both Practical Simplex and Jolteon as the number of processes in the system increases. For the smallest system sizes ($n < 34$), ProSimplex achieves lower throughput than the other two protocols, since the additional cost of its cryptographic operations (used for randomizing and verifying recipient samples) outweighs the benefit of reducing the number of messages exchanged in the probabilistic protocol. However, as the number of processes increases ($n \geq 34$), the probabilistic approach of ProSimplex begins to compensate its additional cryptographic cost, allowing ProSimplex to achieve throughput levels higher than both Practical Simplex and Jolteon. Specifically, Jolteon and Simplex slightly outperform ProSimplex in small systems ($n \leq 10$), with throughput up to 12.47% and 10.86% higher, respectively, depending on the transaction size and the number of transactions per block. As the system scales to $n = 100$ processes, however, ProSimplex consistently maintains higher throughput, ranging from 1,500 to 6,833.33 txs/s, while Jolteon throughput decreases by 9.04% to 12.5% compared to ProSimplex, and Simplex suffers significant drops of 14.63% to 34.68%.

Despite Jolteon's lower message complexity of $O(n)$ in the common case, and its ability to finalize blocks in only 2 communication steps when consecutive leaders are correct - whereas ProSimplex and Practical Simplex require 3 communication steps per block finalization, the three protocols exhibit close throughput results. This is because the dominant overhead lies in the proposal phase, which is similar across all three protocols, as in every iteration the leader must broadcast a large block to all other processes (see Table 7.2). Even if fewer messages are required to finalize a block, all processes must still wait for the leader's proposal to be fully disseminated before a finalization can occur. In practice, the propose phase dictates the protocols' execution because the payload of *propose* messages is much larger compared to the other types of messages. While the network can transmit smaller messages quickly, the broadcast of a large proposal remains the main bottleneck because even a process receives the required messages to finalize a block, it must still wait for the proposal to be disseminated before a finalization can occur. In short, the propose phase may overshadow the differences in message complexity, meaning that even protocols with theoretically fewer exchanged messages end up achieving similar throughput in practice.

The finalization latency results (also detailed in Appendix B, Table B.2) highlight the differences between the number of communication steps required to finalize a block in all protocols. Jolteon consistently exhibits the highest latency values because a block, from the time it is proposed, requires 5 communication steps to reach finalization. Of these, when considering consecutive correct leaders, 2 communication steps correspond to a proposal phase, further increasing the delay before a block is finalized. In contrast, ProSimplex and Practical Simplex achieve lower latency than Jolteon, as blocks can be finalized in as few as 3 communication steps. Practical Simplex always finalizes blocks in 3 steps along the common case, while ProSimplex, due to its probabilistic approach, may occasionally require a second proposal phase, explaining why it presents higher latency than Practical Simplex. In ProSimplex, there is a probability that a process does not observe a probabilistic quorum of *vote* messages required to notarize a block. In such cases, a process must wait for an additional proposal phase to receive a

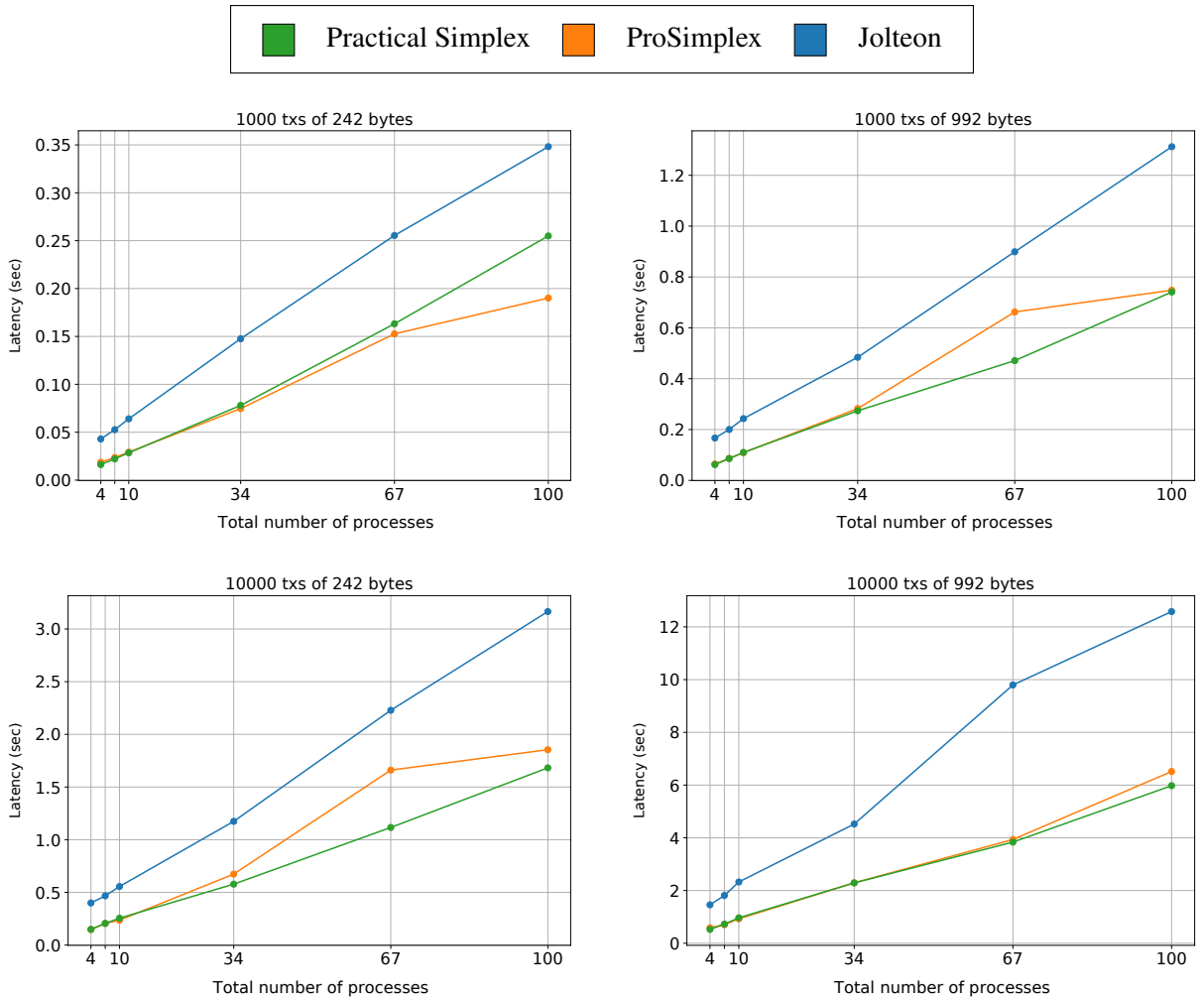

Figure 7.3: Average finalization latency of Practical Simplex, ProSimplex and Jolteon with increasing transaction payload size for blocks with 1000 and 10000 transactions.

valid certificate proving the block's notarization, resulting in a total 4 communication steps for a block finalization. Additionally, a process may also fail to observe a probabilistic quorum of *finalize* messages in a given iteration with a certain probability, in which case it must wait until a probabilistic quorum is eventually observed in a later iteration. Both of these factors increase the average finalization latency in ProSimplex compared to Practical Simplex.

In summary, as expected, ProSimplex scales better than Practical Simplex in terms of throughput for larger system sizes while presenting additional delays for finalization latency due to its probabilistic nature. On the other hand, Jolteon, despite having the lowest message complexity of the three protocols, suffers from higher finalization latency because 5 communication steps must occur between a block proposal and its finalization, whereas in the ProSimplex and Practical Simplex protocols only 3 steps are required in the best-case scenarios. Overall, the throughput results for all the three protocols are constrained by the cost of broadcasting large block proposals which indicates that further techniques for block's dissemination may be required.

Chapter 8

Conclusion

Traditionally, BFT consensus protocols assume an extremely pessimistic approach when dealing with Byzantine adversaries, ensuring safety and liveness properties when Byzantine participants behave completely arbitrarily under worst-case scenarios. This approach can pose inherent challenges in achieving both resource efficiency and high performance especially for BFT systems with a large number of replicas. Some protocols (e.g., PBFT [8]) approach this conflict by opting for low latency and applying message-exchange patterns with quadratic message complexity, while others, (e.g., HotStuff [35]) aim at reducing message complexity at the cost of adding extra communication steps resulting in increased end-to-end response times.

In this thesis, we initially focus on one of BFT consensus protocols, Simplex [9], a recent Byzantine chained consensus protocol that introduces a much simpler approach for ordering blocks of transactions and solves the total order broadcast problem required for implementing blockchains. The main drawback of Simplex is its communication complexity. We address and solve this drawback by designing a new protocol, Practical Simplex. Building on this foundation, we turn our attention to ProBFT [3], a recently proposed probabilistic consensus protocol that achieves sub-quadratic message complexity and optimal good-case latency by relaxing the typical pessimistic assumptions. Although ProBFT provides strong theoretical guarantees, it was never implemented nor extended beyond the single-shot consensus problem. To overcome these limitations, we integrate ProBFT's core mechanisms into Practical Simplex and design ProSimplex, a novel probabilistic chained consensus protocol that preserves the simplicity and block-chaining approach of Practical Simplex while leveraging probabilistic quorums to achieve $O(n\sqrt{n})$ message and communication complexity, and improve scalability.

The Practical Simplex and ProSimplex protocols were implemented in Rust and evaluated experimentally in terms of throughput and block finalization latency. Additionally, we compare both protocols with Jolteon [15], an improved version of HotStuff, that has a linear message complexity in the common case. The experiments demonstrated ProSimplex scales better than Practical Simplex and Jolteon in terms of throughput for larger system sizes while presenting additional delays for finalization latency when compared to Practical Simplex, due to its probabilistic nature. For smaller system sizes, ProSimplex achieves lower throughput than the other two protocols because the additional cost of its cryptographic operations used for randomizing and verifying recipient samples outweighs the benefit of reducing the number of messages exchanged. Jolteon, despite presenting the lowest message complexity of the three protocols, suffers from higher finalization latency caused by the extra communication steps required to finalize a block. In terms of throughput, Jolteon would be expected to be the best performing protocol justified by its better message complexity and its ability to finalize blocks in 2 communication steps when consecutive leaders are correct. Despite this, the three protocols present close throughput

results. This similarity in throughput across the three protocols is explained by the dominant overhead of the proposal phase, where the leader must broadcast a large block of transactions to all processes, making the cost of proposal dissemination outweigh differences in message complexity.

8.1 Future Work

We conclude by enumerating possible improvements to be considered in future work and research. First, further experiments considering Byzantine adversaries would provide a deeper knowledge of of Practical Simplex and ProSimplex's performance and resilience.

Another promising direction is the integration of erasure codes into the proposal phase of both protocols. Erasure coding offers a way to mitigate the proposal phase bottleneck identified in our experiments. Instead of sending the entire block to each process, the leader encodes the block into n coded fragments, such that any subset of k fragments is sufficient to reconstruct the original proposed block. This technique could significantly improve scalability for larger block sizes and make the inherent differences in message complexity and communication steps more evident in the experimental results.

The implementation of the Practical Simplex and ProSimplex protocols can also be improved by exploring block finalization in batches. Instead of writing each block to disk immediately after a deterministic or probabilistic quorum of *finalize* messages is received, blocks could be marked as finalized and only be written do disk once a predefined threshold of marked blocks is reached. This approach would reduce the cost of frequent disk I/O operations and improve the efficiency of message processing.

Finally, the resilience of Practical Simplex and ProSimplex against denial-of-service attacks should be further investigated. In particular, a Byzantine process is capable of continuously sending *request* messages to other processes, forcing correct processes to reply repeatedly. This issue overloads the network and increases the workload of correct processes, significantly degrading performance and a potential solution must implement new mechanisms that prevent redundant and malicious requests from being processed unnecessarily.

Bibliography

[1] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 331–341, 2021.

[2] Tiago Antão, Hasan Heydari, and Alysson Bessani. Towards a Simple and Practical Blockchain Consensus Protocol. *EDCC'25*, 2025.

[3] Diogo Avelãs, Hasan Heydari, Eduardo Alchieri, Tobias Distler, and Alysson Bessani. Probabilistic Byzantine Fault Tolerance. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 170–181, 2024.

[4] Diogo Avelãs, Hasan Heydari, Eduardo Alchieri, Tobias Distler, and Alysson Bessani. Probabilistic Byzantine Fault Tolerance (extended version), 2024.

[5] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. From Byzantine Replication to Blockchain: Consensus is Only the Beginning. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 424–436, 2020.

[6] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[7] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of Byzantine state-machine replication. *Distributed Computing*, pages 1–29, 2024.

[8] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[9] Benjamin Y Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. In *Theory of Cryptography Conference*, pages 452–479, 2023.

[10] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

[11] Tokio Contributors. Tokio - an asynchronous runtime for rust, 2024. Accessed: 2024-12-22.

[12] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing Block Period and Commit Latency in Chain-Based Rotating Leader BFT. In *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 470–482, 2024.

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

[14] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[15] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International conference on financial cryptography and data security*, pages 296–315, 2022.

[16] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[17] Sharon Goldberg, Jan Vcelak, Dimitrios Papadopoulos, and Leonid Reyzin. Verifiable random functions (VRFs). 2018.

[18] Rachid Guerraoui and André Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 168–177, 1996.

[19] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[20] George Kola, Tevfik Kosar, and Miron Livny. Faults in large distributed systems and what we can do about them. In *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference*, pages 442–453, 2005.

[21] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[22] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11):1–11, 2014.

[23] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[25] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 485–500, 2016.

[26] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[27] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Satoshi Nakamoto*, 2008.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[29] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[30] Victor Shoup. Sing a Song of Simplex. In Dan Alistarh, editor, *38th International Symposium on Distributed Computing (DISC 2024)*, volume 319 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:22, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[31] Alberto Sonnino. Hotstuff implementation, 2021. `https://github.com/asonnino/hotstuff` (Accessed: 2025-08-31).

[32] Dalek Cryptography Team. ed25519-dalek: Fast and efficient Ed25519 signing and verification in pure Rust. `https://docs.rs/ed25519-dalek`, 2025. Version 2.2.0; accessed: 2025-09-03.

[33] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[34] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

[35] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

Appendix A

Practical Simplex

Algorithm 2 Practical Simplex – process i.

task `start`()
1: $iter \leftarrow 1$
2: **trigger** `newIteration`($iter$)
task `handleNotarization`($header$, $cert$, txs, h)
3: `notarize`($header$, $cert$, txs)
4: **if** $\neg isTimeout$
5: **broadcast** $\langle$FINALIZE, $h\rangle_i$
6: **send** $\langle$STATE, $header$, $cert\rangle_i$ **to** everyone except i
7: **trigger** `newIteration`($h + 1$)
upon `newIteration`(h)
8: $iter \leftarrow h$
9: `resetTimer`($timer$)
10: $isTimeout \leftarrow false$
11: **if** $i =$ `leader`(h)
12: $b \leftarrow \langle\langle H(b_{|\mathcal{B}|}.header), b_{|\mathcal{B}|}.k + 1, iter, H(txs)\rangle, txs\rangle$
13: **broadcast** $\langle$PROPOSE, $b\rangle_i$
upon expiring $timer$
14: `stopTimer`($timer$)
15: $isTimeout \leftarrow true$
16: **broadcast** $\langle$TIMEOUT, $iter + 1\rangle_i$
upon receiving $\langle$PROPOSE, $b\rangle_j$
17: **pre:** $proposes[b.h] = \perp \wedge j =$ `leader`($b.h$) $\wedge\ b.h = iter$
18: $proposes[iter] \leftarrow b$
19: **if** `extendable`(b) $\wedge\ \neg$*isTimeout*
20: **broadcast** $\langle$VOTE, $\langle b.header\rangle_i\rangle$
upon receiving $\{\langle$VOTE, $\langle header\rangle_j\rangle : j \in Q\} = V$ **from** a quorum Q
21: **if** $header.h > iter$
22: **send** $\langle$REQUEST, $|\mathcal{B}|\rangle_i$ **to** one $j \in Q$
23: **pre:** $header.h = iter \wedge proposes[iter] \neq \perp \wedge$ `extendable`($proposes[iter]$)
24: `handleNotarization`($header, \{\langle header\rangle_j \mid \langle \_, \langle header\rangle_j\rangle \in V\}, proposes[iter].txs, iter$)

Algorithm 2 (continued) Practical Simplex – process i.

upon receiving $\{\langle \text{FINALIZE}, h\rangle_j : j \in Q\}$ **from** a quorum Q
25: **if** $\nexists b \in \mathcal{B}, b.h = h$
26: **send** $\langle \text{REQUEST}, |\mathcal{B}|\rangle_i$ **to** one $j \in Q$
27: **pre:** $\exists b \in \mathcal{B}, b.h = h$
28: `finalize`(h)
upon receiving $\{\langle \text{TIMEOUT}, nextIter\rangle_j : j \in Q\}$ **from** a quorum Q
29: **pre:** $nextIter = iter + 1$
30: **trigger** `newIteration`$(nextIter)$
upon receiving $\langle \text{STATE}, header, cert\rangle_j$
31: **if** $(|\mathcal{B}| < header.k \vee |\mathcal{B}| = header.k \wedge b_{|\mathcal{B}|}.h \neq header.h) \wedge$ `validCert`$(cert, header)$
32: **send** $\langle \text{REQUEST}, |\mathcal{B}|\rangle_i$ **to** j
upon receiving $\langle \text{REQUEST}, l\rangle_j$
33: **if** $l < |\mathcal{B}|$
34: **send** $\langle \text{REPLY}, \{\langle b_k.header, b_k.cert, b_k.txs\rangle\}_{k=l}^{|\mathcal{B}|}\rangle_i$ **to** j
upon receiving $\langle \text{REPLY}, \mathcal{M} = \{\langle b_k.header, b_k.cert, b_k.txs\rangle\}_{k=|\mathcal{B}|}^{l}\rangle_j$
35: **if** $l \geq |\mathcal{B}| \wedge \forall \langle header, cert, txs\rangle \in \mathcal{M}, ($`validCert`$(cert, header) \wedge header.txs = H(txs))$
36: **for** each $\langle header, cert, txs\rangle \in \mathcal{M}$
37: **if** $\nexists b \in \mathcal{B}, b.h = header.h$
38: **if** $proposes[b.h] = \perp$
39: $proposes[b.h] \leftarrow \langle header, txs\rangle$
40: **if** $iter = header.h$
41: `handleNotarization`$(header, cert, txs, header.h)$
42: **else**
43: `notarize`$(header, cert, txs)$

Appendix B

Experimental Results

Table B.1: Average throughput results and percentage differences for Practical Simplex and Jolteon vs. ProSimplex.

Txs	Size (B)	Processes (n)	Jolteon (txs/s)	Practical Simplex (txs/s)	ProSimplex (txs/s)
1000	242	4	59033.33 (+12.47%)	57553.33 (+9.65%)	52486.67
		7	43333.33 (+7.36%)	42273.33 (+4.73%)	40363.33
		10	34116.67 (+5.78%)	32803.33 (+1.71%)	32253.33
		34	13816.67 (+0.46%)	12258.33 (-10.87%)	13753.33
		67	7916.67 (-5.77%)	6486.36 (-22.80%)	8401.67
		100	5700.00 (-9.04%)	4093.33 (-34.68%)	6266.67
10000	242	4	65333.33 (+9.19%)	61900.00 (+3.45%)	59833.33
		7	48333.33 (+6.85%)	46166.67 (+2.06%)	45233.33
		10	40166.67 (+5.79%)	37133.33 (-2.19%)	37966.67
		34	17166.67 (-12.71%)	16833.33 (-14.41%)	19666.67
		67	8666.67 (-11.86%)	9000.00 (-8.47%)	9833.33
		100	6166.67 (-9.76%)	5833.33 (-14.63%)	6833.33
1000	992	4	15866.67 (+10.26%)	15113.33 (+5.03%)	14390.00
		7	11366.67 (+8.01%)	10673.33 (+1.43%)	10523.33
		10	9050.00 (+7.91%)	8430.00 (+0.52%)	8386.67
		34	4100.00 (-9.89%)	3773.33 (-17.07%)	4550.00
		67	2200.00 (-13.04%)	2183.33 (-13.70%)	2530.00
		100	1516.67 (-12.50%)	1383.33 (-20.19%)	1733.33
10000	992	4	16000.00 (+8.60%)	16333.33 (+10.86%)	14733.33
		7	12166.67 (+5.80%)	11700.00 (+1.74%)	11500.00
		10	10000.00 (+1.69%)	9300.00 (-5.42%)	9833.33
		34	4166.67 (-21.87%)	3833.33 (-28.12%)	5333.33
		67	1833.33 (+0.00%)	1166.67 (-36.36%)	1833.33
		100	1333.33 (-11.11%)	1000.00 (-33.33%)	1500.00

Table B.2: Average latency results and percentage differences for Practical Simplex and Jolteon vs. ProSimplex.

Txs	Size (B)	Processes (n)	Jolteon (s)	Practical Simplex (s)	ProSimplex (s)
1000	242	4	0.0429 (+129.97%)	0.0162 (-13.40%)	0.0187
		7	0.0527 (+123.00%)	0.0222 (-6.11%)	0.0236
		10	0.0640 (+119.38%)	0.0285 (-2.20%)	0.0292
		34	0.1476 (+97.70%)	0.0780 (+4.52%)	0.0747
		67	0.2554 (+67.28%)	0.1632 (+6.90%)	0.1527
		100	0.3481 (+83.14%)	0.2549 (+34.11%)	0.1901
10000	242	4	0.3995 (+169.83%)	0.1485 (+0.27%)	0.1481
		7	0.4669 (+128.14%)	0.2054 (+0.38%)	0.2047
		10	0.5556 (+135.52%)	0.2550 (+8.09%)	0.2359
		34	1.1742 (+74.42%)	0.5780 (-14.14%)	0.6732
		67	2.2283 (+34.18%)	1.1160 (-32.80%)	1.6607
		100	3.1656 (+70.74%)	1.6820 (-9.28%)	1.8540
1000	992	4	0.1666 (+158.95%)	0.0619 (-3.78%)	0.0643
		7	0.2001 (+131.91%)	0.0859 (-0.41%)	0.0863
		10	0.2429 (+122.85%)	0.1099 (+0.84%)	0.1090
		34	0.4842 (+71.47%)	0.2737 (-3.07%)	0.2824
		67	0.8991 (+35.75%)	0.4710 (-28.89%)	0.6623
		100	1.3123 (+75.44%)	0.7410 (-0.94%)	0.7480
10000	992	4	1.4588 (+149.14%)	0.5246 (-10.40%)	0.5855
		7	1.8130 (+156.92%)	0.7270 (+3.02%)	0.7057
		10	2.3256 (+150.43%)	0.9661 (+4.04%)	0.9286
		34	4.5234 (+97.36%)	2.2910 (-0.04%)	2.2920
		67	9.7968 (+148.90%)	3.8360 (-2.54%)	3.9360
		100	12.5811 (+93.11%)	5.9780 (-8.24%)	6.5150