Probabilistic Chained Blockchain Consensus

André David dos Santos - 62754
Estudo Orientado
Mestrado em Engenharia Informática
Faculdade de Ciências, Universidade de Lisboa
fc62754@alunos.fc.ul.pt

Abstract

Byzantine consensus protocols typically assume an extremely pessimistic system model when dealing with adversaries and consequently have high message complexity or latency. In practice, adversaries are not as capable as the ones assumed in these protocols, and it may suffice to achieve optimal resilience with high probability. Recently, a new probabilistic consensus protocol, ProBFT, exploits weaker adversaries using probabilistic quorums and verifiable random functions to guarantee safety and liveness properties with high probability. This work will integrate the main mechanisms of ProBFT to implement a practical variation of Byzantine chained consensus protocols and compare it with BFT-SMaRt and other state-of-the-art protocols.

Keywords Byzantine consensus, Optimal resilience, Probabilistic consensus, Probabilistic quorums, Verifiable random functions

1 Introduction

Consensus refers to the problem in which a number of participants in a distributed system attempt to agree on a single common value. A distributed system is bounded to face failures that can end up being adversities when dealing with the consensus problem. We call Byzantine fault tolerance (BFT), the resilience of a distributed system to failures of its participants, in particular Byzantine faults, when attempting to achieve consistency through consensus. In the last years, much research has been conducted to create new practical fault-tolerant algorithms to ensure resilience against Byzantine failures.

Traditionally, Byzantine consensus protocols (e.g., PBFT [5] and HotStuff [22]) assume an extremely pessimistic system model when dealing with Byzantine adversaries to ensure safety and liveness properties. Following this approach typically results in high message complexity (e.g., PBFT) or an increase in the number of communication steps (e.g., HotStuff). However, 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 probabilistic consensus protocol was developed (ProBFT [2]) that guarantees these properties with high probability while requiring sub-quadratic message complexity by relaxing the typical pessimistic assumptions. Despite this, ProBFT has two drawbacks, (1) this protocol has never been implemented or evaluated experimentally and (2) only solves the single-shot consensus problem.

The objective of this thesis is to overcome ProBFT's drawbacks by integrating its core mechanisms to recent Byzantine chained consensus protocols (e.g., Streamlet [7] and Simplex [6]) that introduce a much simpler approach for ordering blocks of transactions and solve the total order broadcast problem required for implementing blockchains. By leveraging ProBFT's main ideas, this project aims to construct a scalable state machine replication and streamlined blockchain consensus protocol inspired by these chained consensus protocols, that eliminates ProBFT's necessity of a view-change sub-protocol. Additionally, this new protocol will be evaluated and compared with two state of the art chained consensus protocols, Streamlet and Simplex, and a traditional state machine replication protocol, BFT-SMaRt [3].

This report is organized as follows: Section 2 presents a background on the problem by tackling and introducing some important concepts. Section 3 describes relevant mechanisms of important protocols which motivated this work, and Section 4 details the implementation of the on-going practical version of Simplex. Section 5 describes future work and Section 6 concludes with a summary of this report.

2 Background

Throughout the 21st century, much development has been made since the first well-known fault tolerant protocols, Paxos [15] and PBFT, with the objective of reducing latency, optimizing resource utilization and achieving simpler protocols that are easier to implement. In the following, some commonly used concepts to categorize the different fault tolerant consensus protocols are presented.

2.1 Faults

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 [13], [17]. Crash faults refer to processes in a system that stop making progress and no longer execute their intended function. 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 [16]. Faulty processes may be controlled by an adversary while non-faulty processes, also called correct processes, are faithful to the description of a protocol.

2.2 State Machine Replication

State machine replication (SMR) refers to a system that provides a replicated service whose state is mirrored across a set of replicas. The term is typically used in the context of systems where clients issue commands and replicas must agree in the order of execution of these commands. In addition to replicas, any number of clients can be also faulty in the respective model.

SMR is defined by the following properties [19]:

- **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.

2.3 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 protocols are used to implement SMR, total order broadcast and blockchains. Nowadays, all of these concepts relate to the the same problem since the primary objective is to reach an agreement among several participants of the distributed system.

The consensus problem defines three properties that must be satisfied [12]:

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

Consensus' protocols goal 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 network 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.

Chained Consensus Protocols. In this category of consensus protocols (e.g., Streamlet [7], HotStuff [22]), the agreements on the state of the system are sequentially chained together. The next agreement depends directly on the previously agreed system state and sequential decisions are cryptographically linked using hashes, forming an inviolable chain. Each decision contains a reference containing the hash of the previous decision, ensuring that any modification to past decisions invalidates the chain and all decisions are final and immutable. In some chained protocols (e.g., Streamlet [7]), it may be necessary to deal with potential forks (divergent chains), requiring mechanisms to reconcile the chain.

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 [2]:

- **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.

Single-shot Consensus. This concept details consensus protocols that execute only one instance of consensus. Each process has an initial value and its objective is to agree on a single value. Single-shot consensus protocols can still move through successive views and each view has a determined leader to avoid liveness issues regarding possible faulty leaders.

2.4 Blockchain

Blockchain is a term popularized by Satoshi Nakamoto in 2008 with the introduction of Bitcoin [18]. 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. At any point, a process's output in a blockchain protocol is its local copy of the blockchain it maintains.

2.5 Communication and Fault Models

In distributed systems and consensus protocols, 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.

This work will only focus on protocols that assume a partially synchronous network model because the main challenge typically associated with consensus and Byzantine failures is to guarantee liveness in an asynchronous scenario, which is the common scenario for practical settings. Asynchronous systems lack any timing guarantees, making it impossible to distinguish between slow and faulty processes. This system model is not appropriate for designing consensus protocols because it makes it impossible to obtain safety and liveness according to the Fischer, Lynch and Paterson (FLP) impossibility [11]. This is the reason why the majority of Byzantine fault tolerant consensus protocols assume partial synchrony [10] where the system behaves asynchronously for some time but eventually becomes synchronous. In a partially synchronous model [6], [2], the system is initially asynchronous but eventually becomes synchronous. Consequently, timing bounds are unknown but eventually hold after an unknown global stabilization time (GST), after which message delays are at most a known upper bound Δ. This is the common system model adopted by Byzantine consensus protocols since it captures real-world scenarios where network delays fluctuate.

2.6 Message and Communication Complexities

Both message and communication complexities are metrics used to evaluate the efficiency of distributed protocols. Message complexity refers to the number of messages exchanged among processes required to achieve consensus and communication complexity is the total number of messages' bits exchanged to finish the protocol. 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.

2.7 Responsiveness, Finalization and Block Times

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 [6], [9]. 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 (finalized decisions) 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 until a protocol considers a timeout. 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 successive proposed values (blocks) being added to the blockchain. Both the finalization time and the block time depend on δ, and/or Δ, depending on the specific protocol.

3 Related Work

This section presents overviews of some Byzantine fault tolerant SMR and blockchain protocols. A simplified comparison of the presented protocols is depicted in Table 1, which denotes the protocol, message complexity, communication complexity, number of communication steps, responsiveness and, block and finalization times for blockchain protocols. As a side note, for the counting of communication steps, client-server communication was not considered.

3.1 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) [5] is a SMR protocol introduced by Miguel Castro and Barbara Liskov in 1999. PBFT was the first practical and scalable 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.

PBFT assumes an asynchronous distributed system where messages in the network can be delayed or lost but not indefinitely since it also assumes fair links to implement reliable channels. The protocol also considers a distributed system with a total of $3f + 1$ processes that tolerates up to f Byzantine faults and public-key signatures of messages.

PBFT follows a leader-based approach and makes progress by changing views. 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 1) is composed by 5 communication phases:

- **Request**. A client sends a request to all replicas.
- **Pre-prepare**. The primary assigns a unique sequence number to the request and sends the request to all backups.

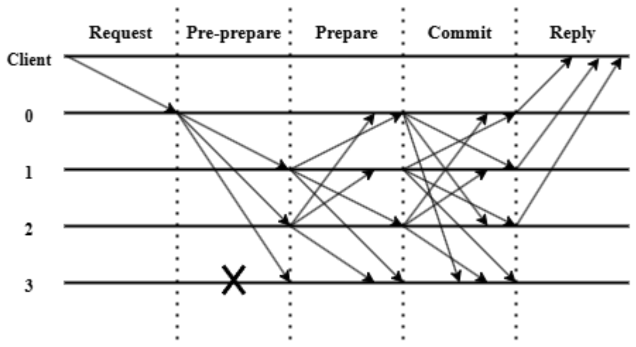

Figure 1. PBFT's normal operation (adapted from [5]). Replica 0 is the *primary* and replica 3 is faulty.

- **Prepare**. Each replica verifies the validity of the request and sequence number, and sends a message to all replicas to avoid being deceived by a Byzantine leader.
- **Commit**. After agreeing on the sequence number of the request, each replica sends a message to all other replicas to agree on the request.
- **Reply**. When replicas agree on the request, each one sends a reply to the client.

The PBFT protocol has a message complexity of $O(n^2)$ since in the normal operation mode, each replica sends messages to all other replicas. 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 collects *view-change* messages from all replicas and broadcasts a *new-view* message containing the collected messages.

During the years, several works looking to improve the original protocol emerged. Zyzzyva [14] improves PBFT's performance by implementing a speculative execution, where each replica speculatively executes a request just after receiving its sequence number from the primary. This work reduces PBFT's latency to 3 communications steps since it considers that the consistent state of the replicas only matters to clients, who will verify if all replicas are on the same state. Spinning [20] increases performance by avoiding Byzantine leaders presence in multiple views and distributing workload among all replicas, which is achieved by changing view in every ordering of requests. MinBFT [21] improves PBFT's resilience and cost by introducing a trusted service that enables a reduction on the number of replicas required for the protocol.

3.2 Streamlet

Overview. Streamlet [7] is a streamlined blockchain protocol introduced by Benjamin Chan and Elaine Shi in 2020. Streamlet's main offer is that it is designed around a minimalist approach to consensus, prioritizing simplicity and ease of implementation. 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.

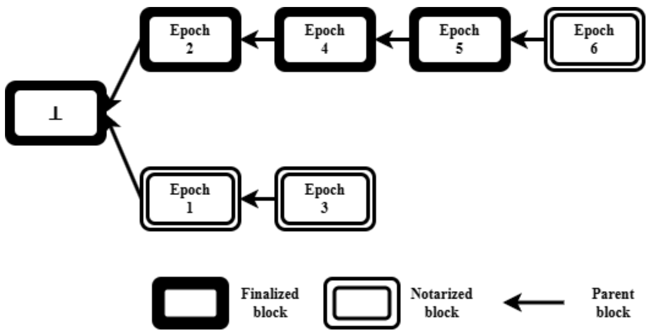

Figure 2. Streamlet's finalization rule (adapted from [7]).

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. Blocks despite containing a hash of its parent block and set of unconfirmed transactions to be executed, also contain 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 call the longest notarized chain, the chain of notarized blocks with the highest length.

Protocol. During each epoch e:

- **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 proposed block to every process.
- **Vote**. During epoch e, processes vote for the first proposal from epoch e's leader if the proposed block extends the longest notarized chain of the process' 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 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 approach, it presents some practical limitations.

3.3 MinStreamlet+

Antão et al. [1] presents the following Streamlet's limitations while also providing an evaluation of a novel protocol that solves them, MinStreamlet+ [1]:

- **Implicit echo**. Streamlet utilizes an implicit echo mechanism in order to guarantee liveness. MinStreamlet+ [1] shows how this mechanism is essential to Streamlet and how removing it would result in possible liveness problems after a Byzantine process becomes a leader for an epoch. 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.
- **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.

The MinStreamlet+ protocol is a variant of Streamlet with improved message complexity, reduced number of steps required to finalize blocks, responsiveness, and increased resilience, effectively solving Streamlet's limitations described next:

- **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 subprotocol 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, the most recent notarized chain is maintained, 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

Overview. Simplex [6] is a protocol introduced by the authors of Streamlet, that also aims 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; 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 3.3.

Simplex achieves responsiveness by advancing epochs only when blocks are notarized. 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). 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 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 block notarization, and a finalization time of 3δ due to the additional finalization phase. Additionally, Simplex achieves a message complexity of $O(n^2)$ since all processes multicast their messages during the three phases and, a communication complexity of $O(n^2)$ because none of the messages' types size is dependent on the number of processes. As a side note, Simplex's *propose* messages must include the entirety of the process' local notarized blockchain and a more realistic communication complexity would be $O(n^2 \times$ size of the notarized blockchain).

Despite Simplex's achievements, it still presents some practical limitations (the practical solutions to these limitations is presented in Section 4):

- **Blockchain in messages**. Simplex's *propose* and *view* messages must include the entirety of a process' 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.
- **Blockchain in messages**. Simplex's *propose* and *view* messages must include the entirety of a process' 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.
- **Dummy blocks**. 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 ProBFT

ProBFT [2] 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 adversaries are not usually as powerful as the ones assumed. In many real-world applications, it is sufficient to assume a static corruption adversary, which chooses a corruption strategy at the beginning of the execution of a consensus instance [2]. ProBFT aims at less pessimistic practical scenarios where ensuring deterministic quorum overlaps can pose inherent challenges in achieving both resource efficiency and high performance but still require less message exchanges and optimal latency. 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.

The protocol considers a distributed system composed by a finite set of n processes and at most f can be subject to Byzantine failures. Each process has 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. 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 [4] responsible for notifying processes when a view changes. Each view consists of three communication steps:

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

To offer resilience against faulty processes 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:

- **VRF_prove**. Given a process i, its private key, and a seed z, VRF_prove returns a sample S_i with the IDs of the different processes selected at random and a proof P_i that allows other processes to verify whether the sample was obtained using this operation.
- **VRF_verify**. Given a public key of a process 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.

ProBFT's message complexity is $O(n\sqrt{n})$ since each correct process after receiving a *propose* or *prepare* message, multicasts another message to a sample of processes of size $o \times q$ ($o \times l\sqrt{n}$). For any view greater than one, a correct process 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 *prepared* messages. Hence, ProBFT's communication complexity is $O(n^2\sqrt{n})$. As a side note, this communication complexity is a consequence of ProBFT being a single-shot consensus protocol that requires a synchronizer to notify processes of view advances and can be improved in a multi-shot implementation where processes are capable of advancing views where there is no need for a synchronizer.

Protocol	*Msg. Complexity*	*Comm. Complexity*	*Comm. Steps*	*Block Time*	*Finalization Time*	*Responsive*
PBFT	$O(n^2)$	$O(n^3)$	3	-	-	✓
Streamlet	$O(n^3)$	$O(n^3)$	-	2Δ	6Δ	×
MinStreamlet+	$O(n^2)$	$O(n^2)$	-	2δ	4δ	✓
Simplex	$O(n^2)$	$O(n^2 \times \textit{blockchain's size})$	-	2δ	3δ	✓
ProBFT	$O(n\sqrt{n})$	$O(n^2\sqrt{n})$	3	-	-	✓
Moonshot	$O(n^2)$	$O(n^2)$	-	δ	3δ	✓

Table 1. Evaluation and comparison of BFT SMR and blockchain protocols.

3.6 MoonShot

Moonshot [9] is a suite of protocols that follow a leader-based propose/vote approach and are the first BFT SMR protocols built under the partially synchronous network model with δ delay between proposals of consecutive correct leaders, achieved by a new optimization called optimistic proposal. This optimization refers to the capacity of a leader process to optimistically extend a block proposed by its predecessor without waiting for it to certified (e.g., observing a quorum of vote messages for a block). In Moonshot, this optimization allows the leader of the next view to propose a new block when it votes for block proposed by a leader of the current view (depicted in Figure 3). In order to finalize/commit a block, Moonshot requires two consecutive views in order to obtain a block certificate for each of the blocks proposed in each view. This results in a finalization time of 3δ: δ for the propose phase, δ for the vote phase and optimistic proposal simultaneously, and δ for processes to vote on the optimistic proposal.

4 Practical Simplex Implementation

A Simplex's implementation is currently being developed with the objective of serving as a base for the future integration of ProBFT's mechanisms and, solve the practical issues present in Simplex listed in Section 3.4. This implementation is being developed using the Rust programming language.[1]

In the following, the modifications made to the base Simplex's protocol to solve the practical problems presented earlier are described.

Timeout mechanism. Taking inspiration from MinStreamlet+'s improvements presented in Section 3.3, a timeout mechanism was implemented to replace *dummy* blocks and avoid the additional memory cost coming from these blocks that do not include any agreed on transactions. If after 3Δ the timer for iteration h expires and a notarization for a block of length h does not happen, a correct process instead of voting for a *dummy* block, multicasts a *timeout* message with the iteration number it wishes to advance to. When a process observes $2f + 1$ signed *timeout* 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.

Missing block's requests. A change of Simplex's base protocol was designed with the of objective of removing the necessity of including the entirety of the local notarized blockchain in *propose* and *view* messages. Removing the local notarized blockchain from these messages sacrifices liveness because messages can be lost before GST, which can result in a process' blockchain lagging behind even after GST since there is no way for a process to synchronize its local state. An example would be a process with a local blockchain $\langle b_0, b_1, ..., b_{h-1}\rangle$, not observing enough *vote* messages for a proposed block b_h before GST because some of them were lost and, eventually after GST, receiving enough *vote* messages for a proposed block $b_h + 1$ that it will never be able to extend because it is missing the notarization for the parent block b_h. To circumvent this issue, a *view* message was changed to only include the last notarized block of the notarized blockchain.

Additionally, this implementation allows a process to detect that it is missing blocks when it observes a notarization for

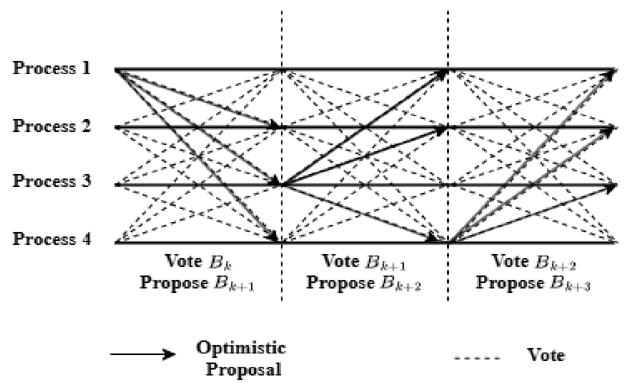
Process 1
Process 2
Process 3
Process 4
Vote B_k Propose B_{k+1}
Vote B_{k+1} Propose B_{k+2}
Vote B_{k+2} Propose B_{k+3}
Optimistic Proposal
Vote

Figure 3. Moonshot's optimistic proposal (adapted from [9]).

[1]https://github.com/AndreSantos0/Probabilistic-Chained-Blockchain-Consensus

a block it is not able to add to the notarized blockchain. This happens when it observes at least *2f + 1* correctly signed *vote* or *finalize* messages or, one *view* message containing a notarized block alongside its corresponding *2f + 1* correctly signed votes. At this point, the process unicasts a *request* to the sender of the last of these messages, containing the number of the highest iteration it has observed a notarization for. When a correct process receives such *request* message, it answers with a *reply* message, containing the requested missing blocks alongside their *2f + 1* correctly signed votes to be verified by the request sender. Although, this change increases the number of messages necessary for processes to update their notarized blockchain when they are lagging behind, it reduces the communication complexity of Simplex to $O(n^2)$.

4.1 Data Structures

The practical Simplex's implementation being developed is composed by the following data structures:

- **Node**. Representation of a system's process host, port and identifier.
- **Environment**. Aggregates the information of all processes in the system.
- **Transaction**. Representation of transaction's data.
- **SimplexMessage**. Enumeration that generalizes all types of messages' structures.
- **SimplexBlock**. Representation of a block's data.
- **NotarizedBlock**. Representation of a notarized block's data alongside the signatures of the *vote* messages that notarized it.
- **SimplexNode**. Initiates and manages communication between processes and contains the protocol's logic.
- **Blockchain**. Represents the local blockchain and includes only notarized blocks.
- **TransactionGenerator**. Simulates and generates the transactions submitted by a client and to be agreed on.

4.2 Communication

System's processes communicate over TCP sockets. In the beginning of the protocol's execution, each process creates a new connection to every other process in the system by opening a new socket. These connections are maintained open throughout the entirety of the protocol's execution. The communication between the processes of the system follows an asynchronous programming model. For each communication opened, a new *task* is created to handle the connection (read incoming messages). The incoming messages are then forwarded to a main *task*, responsible for the protocol's execution, using a multi-producer single-consumer channel shared across *tasks.* Alternatively, there is one more *task* which holds the protocol's timer used to determine if a *timeout* message must be sent and, if that is the case, it signifies the main *task* of said timeout using another channel. It is important to note that *tasks* differ from threads, in the sense that *tasks* do not block a thread's execution during asynchronous operations, allowing for concurrent execution. The implementation of this asynchronous programming model was achieved using the Tokio Rust library [8], a runtime designed for writing reliable and scalable asynchronous applications.

4.3 Methods

Following are the main methods of the data structures that define the protocol's core (SimplexNode and Blockchain):

SimplexNode:

- **execute_protocol()**. Runs for the whole duration of the protocol's execution and creates the main *task* that handles the multiple events of the protocol.
- **get_leader()**. Returns the leader process's identifier for a certain iteration number.
- **propose()**. Broadcasts a *propose* message containing a new block extending the local notarized blockchain.
- **handle_propose()**. Votes for a proposed block following the protocol's vote rule.
- **handle_vote()**. Keeps track of observed *vote* messages, checks for block's notarization or missing blocks, and in the case of a notarization broadcasts a *view* message and a *finalize* message if the timer did not expire.
- **handle_timeout()**. Keeps track of observed *timeout* messages and advances iterations by notifying the main *task* when enough messages are observed.
- **handle_finalize()**. Keeps track of observed *finalize* messages, finalizes blocks by calling Blockchain's *finalize*() method and, deals with garbage collection (explained in 4.4).
- **handle_view()**. Verifies for possible missing blocks and, if that is the case, sends a *request* message.
- **handle_request()**. Answers a *request* message by sending a *reply* message containing the requested missing blocks.
- **handle_reply()**. Checks the messages' signatures of the received missing blocks and adds them to the local blockchain if they are valid.

Blockchain:

- **hash()**. Returns the hash of a block.
- **add_block()**. Adds a block to the blockchain and checks for delayed notarized blocks and delayed finalization events (delays can occur before GST).
- **is_extendable()**. Returns a boolean indicating if a block extends the local blockchain.
- **finalize()**. Finalizes the specified iteration by outputting the respective blocks' representations to a file.

4.4 Garbage Collection

Since the protocol runs indefinitely, it is necessary to clear accumulating data originated from the storage of messages observed during each iteration to avoid implementation bugs that can lead to processes crashing. The implementation keeps four different mappings, one for *propose* messages, one for *vote* signatures, one for *finalize* messages and a final one for *timeout* messages. The *propose* and *finalize* messages can be cleared after a finalization of their respective iteration without sacrificing the implementation's correctness. In the same way, *vote* signatures can be cleared during a finalization when at least $2f + 1$ signatures for a unique, already notarized block have been observed. On the other hand, *timeout* messages can only be cleared when the iteration number contained in them has already been reached. During a finalization, notarized blocks are also outputted to a file to avoid memory issues. More precisely, if a finalization for an iteration h occurs, all blocks notarized during an iteration h or lower are finalized and all blocks notarized during an iteration lower than h are cleared from the process' blockchain. The last notarized block for iteration h despite finalized, must be maintained, for future proposals to be valid since the hash contained in a block is compared to the hash of the last notarized block in the blockchain.

5 Forthcoming Work

5.1 ProBFT's Mechanisms Integration

After the practical Simplex's implementation is fully functional and completed, the next step will be to adapt the practical version of Simplex's chained consensus into a probabilistic consensus protocol by integrating the core mechanisms of ProBFT.

5.2 Performance Evaluation and Comparison

The practical Simplex's implementation as well as its future probabilistic implementation based on ProBFT's mechanisms will be tested using the machines available at LASIGE. Additionally, the throughput and latency of these implementations will be evaluated and compared with Streamlet, Simplex and BFT-SMaRt.[2]

6 Conclusion

This report started by presenting the motivation and an overview of the work being developed under the scope of this thesis. A review on the concepts in the Byzantine fault tolerance consensus area presented throughout this report is made. Following this, related work offers a summary on the main principles of the protocols described and their characteristics and contributions. Additionally, a comparison of the characteristics of each protocol is presented. At last, an overview of the on-going work and how it improves an existent protocol as well as some relevant implementation details are covered.

[2]https://github.com/bft-smart/library

References

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

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

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

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

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

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

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

[8] Tokio Contributors. 2024. Tokio - An asynchronous runtime for Rust. https://tokio.rs Accessed: 2024-12-22.

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

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

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

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

[13] George Kola, Tevfik Kosar, and Miron Livny. 2005. Faults in large distributed systems and what we can do about them. In *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30-September 2, 2005. Proceedings 11*. Springer, 442–453.

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

[15] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

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

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

[18] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Satoshi Nakamoto* (2008).

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

[20] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin one's wheels? Byzantine fault tolerance

with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 135–144.

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

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