Monash University

Distributed Payment Systems

This thesis is presented in partial fulfilment of the requirements for the degree of Master of Information Technology (Honours) at Monash University

By:
Khaled Baqer

Supervisor:
Dr. Ron Steinfeld

Year:
2014

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the work of others has been acknowledged.

Signed by:

Name: Khaled Baqer

Date: 03/07/2014

Acknowledgement

I would like to thank my supervisor, Dr. Ron Steinfeld, for the invaluable feedback and support throughout this project. The feedback obtained, especially regarding security properties and protocols, helped shape this project and added more resilience to the thesis design and implementation. I am also indebted to the many professors that have taught me throughout my two years at Monash; their expertise and support motivated my interest in pursuing challenging research.

Abstract

Financial systems require individuals to have faith in trusted third parties and central authorities, to secure their savings and provide instant access to their accounts. Ultimately, financial systems control assets and individuals have minimal—if any—control over their own assets. This also limits transparency and financial freedoms: Individuals have no control over who can view financial system logs, cannot move assets instantly, and must abide by stringent financial regulations. Moreover, financial systems provide the ability to reverse transactions and reclaim funds; which makes the non-finality of transactions a critical problem when conducting online transactions without establishing trust between entities. Therefore, one of the main aims of this research is to design an alternative to the existing trust-based financial systems.

This research focuses on consensus algorithms to provide the alternative solution for our system architecture, focusing primarily on Byzantine Fault Tolerance (BFT) algorithms. BFT algorithms provide the necessary infrastructure to design resilient protocols that operate in a hostile environment, where other nodes cannot be trusted. In order to understand the perils of existing research—to adopt its strengths and avoid its weaknesses—we first explore Bitcoin, which is a prominent financial system. We define the problems in Bitcoin that we aim to avoid (with regards to privacy), and focus our research on critical problems that we aim to resolve (global ledger consensus).

In this thesis, we provide detailed analysis of our findings and implementation, for researchers to be able to replicate the research if necessary. We also evaluate the research to provide an overview of the feasibility of our implementation. The final outcome is a direct result of the research aims: A design for distributed payment systems that cannot be controlled or shutdown by a single entity, while actively defending against abnormal behaviour and network attacks through a cooperative consensus process. We created a distributed payment system, unregulated by a central authority, that provides almost instantaneous, final, and irreversible online transactions, without the need to establish trust.

Table of Contents

List of Algorithms

List of Figures

1 Introduction

We discuss in this chapter the research background and context. We provide a discussion regarding the motivation for conducting this research, highlighting problems that require non-trivial solutions, which we aim to explore in this research.

1.1 Problem

Financial systems require individuals to have faith in trusted third parties and central authorities to secure their savings and provide instant access to their accounts. This arrangement proves viable until some governments imposed heavy regulations on their financial systems as a result of financial crises in recent years. Moreover, existing electronic payment systems do not provide a solution. Whether it is PayPal or Visa, transaction fees are imposed on individuals and businesses, and delays of processing payments are not a rare occurrence. It is important to note that the aforementioned trust requirement between customers and banks results in delayed transactions, and requires third-parties and central authorities, that are heavily regulated, to establish trust. A critical issue we try to overcome in this research is the non-finality of transactions; transactions can be disputed and funds reclaimed even after transactions are confirmed. Furthermore, these heavily regulated and centralised entities control access to customers' funds and can deny access to a customer. This reflects a bitter reality: individuals have minimal, if any, control over their own financial assets. Although alternative solutions have been proposed, this research explains the inefficiencies and unreliable assumptions surrounding such systems. The problems regarding heavy centralisation and regulation, risky trust requirements, as well as delayed transactions and their non-finality highlight a need for a practical solution to create an unregulated alternative system, that avoids the aforementioned perils of current financial systems.

1.2 Aim

This research aims to create the necessary architecture for a distributed payment system, unregulated by any central authority, which requires the cooperation of participants to agree on the state of the system. It is important to avoid the perils of existing alternatives by building an efficient system that does not require constant consumption of computational power and energy. Therefore, this research aims to create a reliable and distributed payment system with the majority of nodes controlled by non-malicious users. Although this system is autonomous and no central authority should be able to control the entire system, participation for reaching consensus will be limited to preselected quorums.

The main focus of this research is to identify possible architectures for deploying distributed payment systems, evaluate security and feasibility, and compare the designed architectures to existing systems. We aim to produce a practical implementation that satisfies the main research goals (as identified in the next subsection), providing full technical information regarding the implementation (for implementers to replicate this research), and evaluate the implementation and compare it to existing alternatives (to highlight the feasibility of our implementation).

1.3 Goals

To achieve the aforementioned research aims, specific research goals have been set to create a resilient system that can scale to work with or replace existing unregulated financial systems. The research goals are as follows:

- Create a global ledger within distributed systems through consensus. The ledger is unregulated by a central authority.
- The process of consensus must require cooperation to reach agreement without the need to expend constant and continuous energy to maintain consensus.
- Deploy cryptographic techniques to provide the necessary mechanisms to conduct transactions and interact with the global ledger.
- Deploy network security techniques to protect against malicious users and abnormal usage patterns.
- Provide fast and final transactions that maintain an acceptable level of user privacy.

1.4 Thesis Structure

As was highlighted in this chapter, an alternative to the existing centralised financial systems is a necessary requirement to conduct unregulated and irreversible online transactions. We outline below the thesis structure: Part I is the theoretical context of the thesis, where we explain the rationale of our design. Part II is the practical implementation, where we provide detailed technical information regarding reproducing the entire research.

Part I: Theory

- **Chapter 2**: In *Literature Review*, we explore the existing alternative financial systems and their problems. We also highlight possible solutions from the literature, and define theoretical and fundamental concepts used throughout the research.
- **Chapter 3**: In *System Model*, we provide our system assumptions, as well as information regarding our system environment.
- **Chapter 4**: In *Threat Model*, we explore prominent threats to our system. We outline attack vectors that must be contained in our implementation.
- **Chapter 5**: In *System Design*, we provide the high-level theoretical view of our system, explain our system design, and provide reasons for design decisions.
- **Chapter 6**: In *Protocol*, we discuss in more details the high-level overview of the system provided in the previous chapter. We also include flow charts which simplify the understanding of subsequent practical implementation chapters.

Part II: Practical Implementation

- **Chapter 7**: In *Implementation*, we provide detailed information regarding how we programmed the entire system from scratch using Python. This chapter relies heavily on concepts explained in previous chapters.
- **Chapter 8**: In *Evaluation*, we aim to provide an analysis of system performance and costs. This information is required for implementers and researchers to analyse the feasibility of our research.

- **Chapter 9**: In *Challenges and Limitations*, we aim to outline the limitations and system improvements, that can be added to the current implementation to provide a higher level of security and resilience.
- **Chapter 10**: In *Conclusion*, we provide a summary of our research, highlighting our findings and results, and our main implementation components.

2 Literature Review

In this chapter, we describe Bitcoin—a prominent alternative financial system—and discuss problems in Bitcoin's current implementation that motivated this research. Moreover, we describe concepts used by Bitcoin that employ cryptography to establish identities and provide methods for transaction authentication, without the need for deploying a Public Key Infrastructure (PKI).

We also discuss possible alternatives and solutions to Bitcoin's problems, outlining fundamental concepts obtained from the literature on distributed systems. We explore consensus algorithms, focusing on Byzantine Fault Tolerance (BFT), which influenced our research implementation.

2.1 Bitcoin

In late 2008, *Satoshi Nakamoto*[1] introduced Bitcoin as the world's first decentralised peer-to-peer (P2P) cryptocurrency [56], and the first software implementation was released in early 2009. A cryptocurrency is an electronic currency based on cryptography to regulate how the currency is created as well as how users interact with the system. Cryptography is used to generate transactions: Alice signs a transaction to Bob that includes one of Bob's public keys. Alice proves her ownership of the required funds to pay Bob by accumulating in her wallet a list of signatures from other individuals or entities signing transactions to public keys belonging to Alice. It is important to note that public keys are not physically exchanged; transactions include the hash of the recipient's public key (and not the public key itself), which she advertises publicly, and represents one of her *addresses* (many key pairs can be generated, allowing for different pseudonymous identities). Entities wishing to spend their funds must prove that the hash of one of their public keys generates the same hash of the pseudonymous identity attached to the transaction [17, 23]. This effectively enables the authentication of transactions to be conducted without central authorities to verify identities (authentication is performed through checking public keys and their hash digests). Figure 1 shows the outline of conducting transactions in Bitcoin.

To replace the need for a central authority that maintains financial records, Bitcoin aims to establish global consensus through a process called Proof-of-Work (PoW)[2] based on the concept proposed in Hashcash [2]. Powerful nodes[3] gather transactions, which had propagated through the P2P network, and proceed to verify them and perform expensive cryptographic computations using hash functions. These nodes produce the required result of a hash digest with a specific collision: a prefix with a pre-defined number of zeros in the hash resultant. Nodes must inject random values into the hash function to produce the required collision. This process takes time and consumes energy, thus proving that the node has performed the required computations to reach the desired result. Moreover, the resultant hash digest must also incorporate all previously

[1] Widely agreed to be a pseudonym, *Nakamoto* will be used to refer to an entity rather than a person.

[2] Appendix A1 contains Python code, created by the author, for a simple demonstration of PoW.

[3] Power is defined as the node's ability to afford expensive energy consumption to execute computationally intensive operations.

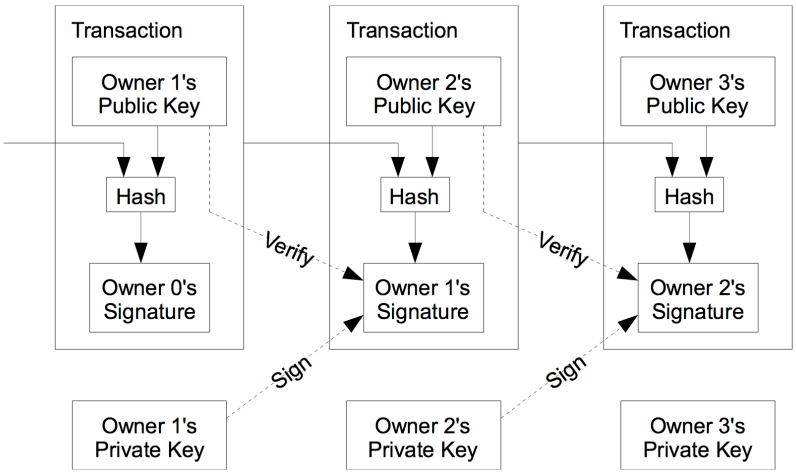

Figure 1: Bitcoin transactions [56]

produced digests including the first digest. Each of the produced hash results is called a *block*, and the chain of blocks hashed together one after the other—including the first block called the *genesis* block—is called the *blockchain*. Figure 2 shows the process of hashing and chaining blocks. Powerful nodes are also referred to as *Miners* as a metaphor for real-world miners who exert substantial effort to mine precious metals. Bitcoin Miners create Bitcoin's currency, also called *Bitcoin*, through the generation of blocks to claim rewards [38].

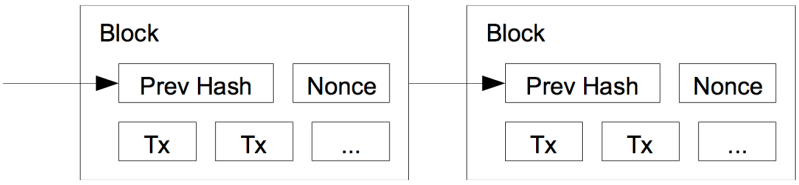

Figure 2: Process of hashing and chaining blocks [56]

Bitcoin defines time in the context of the blockchain, and transaction precedence and timestamps are irrelevant, which is effective in preventing *double-spending* attacks. A double-spending attack happens when Alice tries to pay Bob and Charlie with the same Bitcoins. If Alice broadcasts two different transactions using the same funds, only one of them will be incorporated into the blockchain. Miners will invalidate the other transaction, regardless of timestamp and precedence, during transaction verification after noticing that a transaction uses funds that had already been spent [33, 34, 60]. Bob and Charlie should not accept payments unless a transaction is included in the blockchain, otherwise double-spending attempts can be trivially successful [5,33,34]. This is the most significant contribution by Nakamoto [56, 57]: The blockchain PoW prevents double-spending attacks and eliminates the need for a trusted third party. Moreover, the design relies on the monetary incentive, given to Miners in the form of Bitcoins for producing difficult

PoW, to verify transactions and prevent attacks by making it difficult to create alternative blocks and alter consensus [1, 38].

Therefore, Bitcoin assumes and requires that the majority of the network's computational power is owned by non-malicious nodes, to produce and maintain global consensus. Bitcoin requires that the number of *honest* (non-malicious) nodes n is at least $2f+1$ where f is the number of malicious nodes. Thus, Bitcoin requires at least $\frac{n}{2}$ nodes to control the network, which defines a majority. However, the consensus spectrum, as defined by Bitcoin, is all the existing global computational power. The Bitcoin network, although it has witnessed astronomical increases in hashing power (also referred to as hash-rate) [12, 38], is not representative of all the global computational power in existence. As discussed by Laurie in [47], until we experience all the computational power in the world, continuously and forever, we cannot conclude that we have reached a global consensus. Therefore, although we can reach a consensus, we can never be sure that it is the correct consensus based on how Bitcoin was designed.

Bitcoin's assumption about computational power opens the system to attacks by malicious users who can outperform honest nodes. Attackers need control of computational power that is larger than all the honest nodes' computational power combined. This means that even a single powerful entity can violate Bitcoin's requirement regarding the majority of computational power. An attacker would be able to generate blocks to collect block rewards or invalidate funds the attacker had previously spent. This attack is commonly referred to as the 51% attack; reflecting Bitcoin's need for a majority of honest nodes to operate correctly [17,33,34]. Ultimately, the attacker decides legitimacy, validation, and—as defined by Bitcoin—controls time and can predict the future. To exacerbate the problem, the Bitcoin protocol is incapable of automatically detecting double-spending attacks or an alternative blockchain, also called a *fork*, leaving the system vulnerable to such attacks.

Although the original Bitcoin protocol aimed to establish equal participation, based on the concept of "one-CPU-one-vote" [56, 57], this is hardly the case anymore. Customised hardware components, such as ASIC Miner chips, substantially increased the network hash-rate, with shipments frequently delayed because of excessive and unexpected demand [38]. If a node does not own powerful equipment to generate hash digests rapidly, it is merely wasting power. Practically, a non-powerful node's CPU-vote is inessential for consensus since it will never equal votes of the powerful nodes in terms of computational capabilities. Moreover, Miners work collectively to share resources and increase their chances of mining a block and claiming block rewards; this collective work is called a *Mining Pool*. Therefore, it is reasonable to conclude that Bitcoin has transformed into a distributed system—rather than a fully decentralised P2P system as originally envisioned—operated by powerful Mining Pools that control the global consensus. Figure 3 shows the distribution of hashing power in Bitcoin's network[4]; demonstrating that Bitcoin's consensus is maintained by distributed nodes.

It is important to note that although Bitcoin's consensus protocol may create an agreement on the state of the system, Bitcoin's PoW does not include a cooperative process by a majority of the network; merely an acceptance. Nodes accept the longest blockchain as the legitimate transaction log with the assumption that a longer blockchain reflects more work performed by honest nodes [12, 17, 56]. Therefore, since no cooperation or interaction is required to confirm outcomes, a powerful entity can produce consensus on its own, and impose its consensus on the network. This is equivalent to having a single powerful node mining the blockchain forever. In other words, Bitcoin's PoW is hardly a consensus at all; it is an effective authoritarian imposition protocol.

[4]Obtained from *blockchain.info*

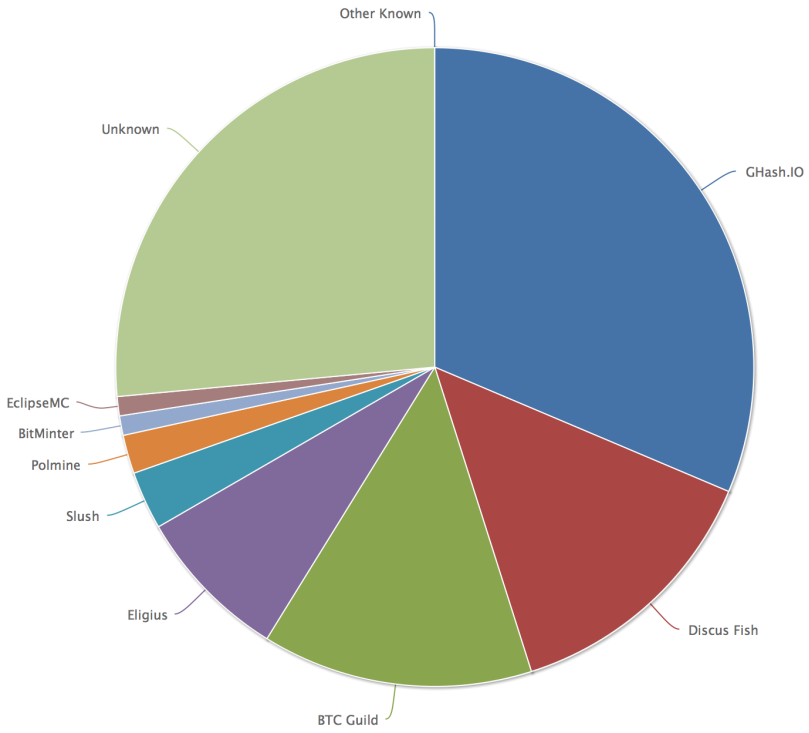

Figure 3: Bitcoin's distributed mining pools

2.2 Laurie's Blueprint

Even before the inception of Bitcoin, Laurie expressed skepticism regarding the efficacy of PoW deployed to fight spam as implemented by Hashcash [49]. Laurie criticised Bitcoin for its use of PoW to achieve consensus, and his ideas heavily influence this research. Laurie published a paper detailing that decentralised currencies are impractical and are difficult to implement in a paper titled *Decentralised Currencies Are Probably Impossible* [47]. *An Efficient Distributed Currency* [48], published a few weeks after the aforementioned paper, aimed to sketch an outline for a distributed currency. Laurie's main ideas can be summarised as follows:

1. The consensus is based on a *snapshot*: the state of the distributed system including coins and ownership. This reflects Lamport's definition of replication consensus that relies on distributed snapshots [11].

2. Consecutive snapshots are hashed into a Merkle tree.

3. Control server membership, whereby the distributed servers are known. Server membership can be controlled by exerting computational effort, in the same manner as PoW.

4. To verify the state of coins and transactions, *clients* can directly query the state snapshot. Clients can also require the entire snapshot, not just the result of a specific query, to verify for themselves. Clients may also query other servers if they are skeptical that any server is trustworthy.

5. Servers can only confirm transactions when consensus with all other servers is established.

Our system design is influenced by Laurie's criticism of Bitcoin and the aforementioned blueprint. However, we diverge from the blueprint to provide a practical implementation that fits our research aims and goals, within a reasonable timeframe. The rest of the chapter is structured to synthesise concepts found in the literature to provide a practical solution for cooperative consensus, seeking an alternative to Bitcoin's non-cooperative and computationally expensive PoW consensus.

2.3 Distributed Systems Properties

We now explore properties of distributed systems that we require in our system architecture, with regards to network communication (synchrony), protocol execution requirements (safety and liveness), and failure models we aim to prevent.

2.3.1 Synchrony

Our system requires asynchronous communication between nodes. In an asynchronous model, processes exchange messages with unbounded delay, which is a practical approach since nodes in our system architecture will communicate over the Internet and we cannot make assumptions regarding network latency and performance [6, 28]. Moreover, setting time constraints is risky, since adversaries can attack the network and delay communication, thereby rendering synchrony assumptions invalid and impractical [16, 65]. However, for consensus to work successfully using distributed servers, we need to enforce a partial synchrony requirement as discussed in later sections.

2.3.2 Safety and Liveness

We also require that our system provides both safety and liveness. Safety can be viewed as the lack of abnormal or unexpected behaviour in the system. Liveness is the property referring to the *eventuality* of system results; a result must eventually be obtained, and the system does not halt without producing a result [6, 28].

2.3.3 Failure Models

In order to build resilience into our system architecture, failure models must be accounted for. We employ the definitions used in the literature as follows: *Fault* is the defect in systems that may cause an error. An *error* may eventually lead to a *failure*, whereby a system diverges from expected behaviour [28, 64].

Systems can be categorised into three types based on the system's approach of handling failures, called a system's failure model. *Crash-failure* model considers a failed process to be one that simply halts and is not functional. *Fail-stop* model assumes that processes can fail, yet failure is easily detected by other nodes in the network, and the network behaviour is adjusted to account for the detected failure [28,61,65]. The aforementioned models do not consider malicious behaviour or an adversary that can take control over network nodes. Therefore, we aim to focus on the third type of failure classification, *Byzantine* model, where components can exert failure behaviour, and may act maliciously. We discuss this model and the problem it entails in later sections.

2.4 Consensus

In this section, we aim to define consensus in general, and distributed consensus in particular. We outline the basis for consensus algorithms, which is used as a platform for Byzantine algorithms that apply cryptography. Before discussing Byzantine algorithms, we cover cryptographic properties that are required to be satisfied to reach valid Byzantine consensus results, and how a distributed system can be made resilient to malicious behaviour. We then discuss how Byzantine algorithms deploy cryptography to non-Byzantine algorithms to satisfy the identified cryptographic properties. Moreover, we provide analysis of security features provided through cryptography and analyse system performance of consensus algorithms.

2.4.1 Defining Consensus

Although consensus can be viewed differently regarding the initial state of the system, the main idea is to reach agreement on the final outcome. Some studies view the *consensus* process as *every* component with an initial value exchanging messages to agree on a single value [31, 32]. Other studies refer to consensus as agreement: A single component proposes a value that must be eventually the same value for every component including the proposer [15, 16]. However, regardless of the initial system state, certain properties must be satisfied to reach consensus [7, 15, 16, 18]:

i. **Validity:** Every final outcome must be a proposal that is valid and non-malicious.

ii. **Agreement:** All components agree on a single outcome.

iii. **Termination:** The process must end eventually and cannot run forever (liveness).

iv. **Efficiency:** The message complexity (communication cost) is bounded.

The first two properties address system *safety* to require the system to produce globally accepted values and avoid abnormal and unexpected outcomes. The third property addresses the system's *liveness* requirement to eventually produce an outcome. Without termination, the system would be impractical since it may stall without producing any results [16]. The fourth property addresses a network with malicious components; the number of messages and their complexity may increase when components are persistent in sending conflicting commands [7].

Although the concept is simple to define, reaching consensus in a distributed system is a non-trivial process [45, 59]. Distributed consensus is the process of reaching an agreement among an assembly, referred to as a quorum. This agreement is established by the majority of the participants deciding an outcome that is adopted as the quorum's consensus value [63, 67]. A majority is required for the decision to render contradicting messages invalid. In the following sections, we discuss required background for consensus in distributed systems as well as the role of faulty and malicious components.

2.4.2 Byzantine Generals' Problem

The Byzantine model was first discussed in [30] as the *Two Generals' Paradox*. However, the model obtained its name from [45, 59] where the problem was hypothesised as the difficulty to reach agreement between Byzantine army generals. In this model—commonly referred to as the *Byzantine Generals' Problem*—a node may exert normal failure behaviour (covered in crash-failure and fail-stop models), but behaves so with a malicious intent to cause system confusion. The idea of faulty components attracted much research due to the critical impact faulty components can produce within a system [45, 59]. Faulty components are generally considered to

be either malfunctioning or malicious components. Components propagate errors through the system, simply halt or crash, or are forced to broadcast errors for a malicious purpose or by an adversary, that can reflect the first two possibilities.

It is important to understand the problem which explains the difficulty of consensus in the presence of malicious nodes. Therefore, we explain the Byzantine Generals' Problem in more detail: Byzantine generals, in the existence of uncertainty about the faithfulness of generals and the unreliability of messengers, try to reach consensus to attack an enemy. When there are only three generals, it is proven that consensus is impossible to achieve [24, 45, 59]. Hence, unreliable and unauthenticated messages in Byzantine environments require the total number of nodes in the network n to be more than $2f + 1$, where f is the number of Byzantine components, and $n > 3$. In a purely P2P network, it is difficult to cooperate interactively with all nodes since reaching consensus regarding which nodes are connected to the network (unknown participants) is a consensus problem in itself [55]. This limits the practicality of involving every node in the network in the consensus process, and highlights the need to carefully consider the ratio of the total number of nodes in the system n, versus Byzantine components f to maintain consensus.

Thus, we can view the consensus problem in a distributed environment with Byzantine components, as the problem of agreeing on a certain value or command by a majority number of non-faulty processes in the existence of Byzantine components [16]. Furthermore, Byzantine components, through sending conflicting commands, aim to prevent the system from reaching agreement and consume valuable resources available on non-faulty nodes, which is a process called *starvation* [53]. We design our system architecture to reach consensus with the existence of Byzantine components to prevent starvation and thwart malicious behaviour.

2.4.3 FLP Impossibility Result

Research into consensus faced a discovery that halted progress until researches obtained a better grasp regarding its implications. The *celebrated impossibility result* [26] is known by its authors' initials as the FLP *impossibility* result, or simply FLP. FLP indicates that it is impossible to reach consensus in a purely asynchronous environment, even if one component is Byzantine, without incorporating randomisation, changing the system model, or making other practical assumptions about the system environment. Moreover, FLP stresses that consensus is impossible to achieve for the lack of—at the time—a mechanism to determine if processes have crashed or are simply delayed due to network performance [16, 31]. Therefore, subsequent algorithms focused on avoiding FLP, a process commonly referred to in the literature as *circumventing* FLP, and gave birth to a rich field of research into fault tolerance.

2.5 Fault Tolerance

In this section, we first examine practical approaches that have been selected from the literature to provide mechanisms for fault tolerance. The ideas presented in this section are concerned with replicating reality, and generating system snapshots reflect Laurie's blueprint [47, 48]. Moreover, the need to detect failures is heavily relied upon in consensus algorithms we discuss in later sections. Therefore, there is a need to examine these concepts before exploring the details of the algorithms. Cooperative consensus for a replicated global log is fundamental to our system architecture. This section examines the high-level approaches that attempt to establish this global view.

2.5.1 State Machine Approach

Lamport defined time and synchrony in [39], providing the fundamentals for the *state machine approach* [39, 61]. The state machine approach is a replication and management mechanism to govern system components' interaction. Replication is essential to achieve consensus in a distributed system. By agreeing to replicate a certain version of reality, other faulty or malicious components that try to alter this reality will fail due to the agreement reached by non-faulty components [28, 59, 61]. Therefore, system nodes agree to replicate the same version of reality, which is also referred to as a *snapshot*, that governs consensus in a distributed environment [11].

As defined by the seminal work regarding state machine replication in [61], a client requests the execution of commands. The request contains all the necessary information required to validate the request and perform the necessary command. Moreover, adapting to asynchronous environments and closely reflecting work initiated by [39] with regards to logical time, the outcomes of the executions reflect the order of received commands, rather than abiding by user-specified timestamps or a globally synchronised physical clock. Therefore, we can summarise the benefits of running state machine replication as follows [36]:

1. Managing a replicated log in distributed systems; ultimately creating a global snapshot that represents the global view [11].

2. Provide a disaster recovery mechanism for hosts that experience crash-failures and need to synchronise with a global view.

3. Use *cooperative consensus* to replicate the state machine, whereby servers work together to agree on the state of the network. The process accounts for an asynchronous environment where messages can be arbitrarily delayed or lost during transmission.

2.5.2 Unreliable Failure Detectors

Failure detectors, first introduced by [10], are used to predict the performance of other nodes, in order to detect which components have crashed (and circumvent FLP). Failure detectors are often referred to as *unreliable* failure detectors, since the information they predict can include false positive indicators [16, 28, 32]. The prediction process involves asking remote nodes to perform certain actions, and in many cases this is performed using a network heartbeat as implemented in [32]. The failure detector works based on suspicion: If a heartbeat is not received after a certain amount of time, failure detectors suspect that the remote node has failed. If the remote node resumes communication and replies with a heartbeat message, then the failure detector reverts to a non-suspicious state. Moreover, the timeout can be increased to account for possible network delays, and the timeout can be dynamically adjusted [10]. This process involves enforcing system timeouts, effectively requiring some synchrony assumptions [7, 20].

Furthermore, it is important to note that unreliable failure detectors are not highly effective to predict failures in the presence of Byzantine components [6]. This difficulty stems from the nature of Byzantine behaviour; components can send false information regarding their state, delay messages, or completely withhold information. In the best-case scenario, failure detectors are shown to be capable of detecting only subsets of Byzantine components [3, 16, 35]. This malicious behaviour renders the prediction process ineffective. However, we mention unreliable failure detectors since the basis of consensus algorithms, that we discuss in a later section, assumes non-Byzantine components and works in partially synchronous environments.

2.5.3 Byzantine Fault Tolerance (BFT)

System resilience is described as Byzantine Fault-Tolerance (BFT) and measured as a system's capability to withstand and defend against Byzantine behaviour, through a majority of non-Byzantine components [9, 37, 44, 51]. For example, the Byzantine Generals' Problem requires that a certain number of non-malicious nodes n be larger than faulty Byzantine nodes f. More specifically, in the presence of Byzantine nodes, and with the lack of message authentication, consensus can be reached only when $n \geq 2f + 1$ [4, 29, 42, 45, 59]. Therefore, the number of Byzantine nodes must be less than half the total nodes in the network. We will revisit this discussion in the *Performance Analysis* subsection.

2.6 Consensus Algorithms

In this section, we examine consensus algorithms that create the foundation that subsequent algorithms have used as basis for their implementations. We explore first the fundamental protocol (Paxos), and then identify enhancements found in the literature that provide security, enhance performance, and create a more practical solution that fits well into our system architecture.

2.6.1 Paxos: Foundation for Consensus

Paxos algorithm, introduced by Lamport in [40], is a consensus algorithm that accounts for component failure (crash-failure model) but not Byzantine components [15]. Paxos and Chandra-Toueg algorithms [10] are similar in their approach and steps necessary for establishing consensus [32, 50]. Although the naming for the elected node is different—Paxos calls the node *proposer*, while Chandra-Toueg calls the node *coordinator*—the role of the elected node is the same: To gather server votes and coordinate consensus operations [32]. Both algorithms deploy failure detectors (discussed in §2.5.2) as their approach to circumvent FLP (discussed in §2.4.3)—and hence work in the fail-stop failure model—and state machine replication to create a global view (discussed in §2.5.1). Moreover, Paxos is proven to be more efficient, particularly in the critical case when a leader (proposer) crashes [32]. Therefore, we examine Paxos since it is extensively discussed in the research, and for its practical applications and widespread use in software implementations.

In Paxos, systems communicate through sending messages and the designated roles are *proposers*, *acceptors*, and *learners*. A proposer proposes *values*; numbered messages that aim to be selected as the consensus value. Acceptors learn of proposals and follow certain principles in order to successfully reach consensus. Paxos maintains progress through the selection of a proposer (leader) by utilising the failure detector Ω [36, 41, 46]. If the proposer crashes, and this crash has been correctly identified by the failure detector, the failure detector runs another round for the proposer selection process. The failure detector is used by nodes in the network to send regular heartbeat messages (network packets, for example ICMP messages) to other nodes in the network. If the node does not receive a reply to its heartbeat query after a certain period of time, its failure detector suspects that the remote node has failed (as discussed in §2.5.2). The node can act as the proposer and assume the role of the leader. If conflicting messages are sent in the network by different nodes that assume they are the proposer, then the consensus algorithm is aborted and the protocol is reinitiated (*abortable* consensus) [6]. A simplified outline of the algorithm is provided below, and a diagram of Paxos operations is shown in Figure 4 [32, 36, 41, 46, 50, 52]:

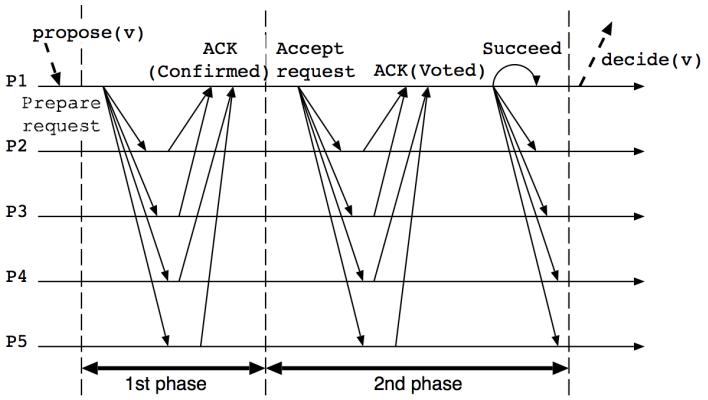

Figure 4: Paxos operations [32]

PHASE 1

1. **INITIATE**. Fault detector Ω elects a leader using the mechanism for selecting a leader discussed earlier in this section. The leader is called a *proposer*.

2. **PROPOSE**. A proposer sends a *prepare request*, with a proposal value α, numbered i and sends α_i to all nodes, requesting the *acceptors* to respond with either of the following responses:

   (a) A promise not to accept proposals numbered k where $k < i$.

   (b) The proposal β_k with the highest number less than i that the acceptor has already accepted.

3. **INITIAL VOTE.** The proposer waits for responses that represent the view of the majority of the nodes in the network. A majority view is defined as replies from $2f + 1$ nodes; f is the number of faulty components.

PHASE 2

4. **ACCEPT**. If a majority view is obtained that matches (a), then the proposer broadcasts an accept message with the initial proposed value α_i, otherwise the proposer broadcasts the network-accepted value β_k received in (b). This is accomplished by sending an *accept request* to a group of acceptors with the aim that a final value is accepted and a consensus is reached.

5. **FINAL VOTE.** The proposer waits for $2f + 1$ replies from acceptors that confirm the acceptance of the value attached to the accept request in step 3.

6. **COMMIT**. When a final outcome is agreed upon, *learners* must be informed of the final value. Learners will ultimately decide on the protocol outcome based on acceptors' replies, to commit commands to the state machine.

7. **END**. The majority value returned to the proposer in FINAL VOTE, and committed by learners in COMMIT, is the final value of the consensus protocol in that round.

2.6.2 Cryptographic Properties

In this subsection, we discuss cryptographic techniques that can limit the effects of Byzantine components and enhance resilience of consensus algorithms.

By utilising asymmetric cryptography—known as Public-Key cryptography—an algorithm can require messages to be signed by system components. This requirement leads to the verification of messages; providing both the authenticity of the message by establishing the identity of the sender, and the integrity of the message contents. Therefore, the system can limit the behaviour of Byzantine components; using message authentication, and assuming that the deployed cryptography is effective, Byzantine components cannot forge messages or alter contents. Message authentication can be established through using an authenticator, which contains data created by one entity, and can be easily verified by another entity [21, 59]. In other words, message hash digests can be used as redundant data which can be signed by the sender to prove authenticity [25,54,62]. Thus, malicious components can be easily identified and isolated to limit their capabilities [21]. This process can limit possible Denial of Service (DoS) attacks based on quick signature verifications to discard unwanted messages. We now list cryptographic and security properties, identified in [29], and can be viewed as basic requirements for any algorithm that must account for—and effectively contain—Byzantine components:

a. Signatures must not be commutative: $\{\{v\}p\}q \neq \{\{v\}q\}p$, where $\{x\}i$ means a message x is signed by entity i.

b. Verify signature sequence, by assigning redundant data to the signatures.

c. Effectively detect and prevent replay attacks. This requires preventing attacks where a component, whether maliciously or due to system failure, uses obsolete data and resends the message to the network. Therefore, *freshness* data, whereby a user proves that the message is recent must be required through timestamps, random nonces, or counters.

d. Redundant data must be attached to messages to effectively establish authenticity. This point reflects the aforementioned concept of message authentication, using authenticators and redundant data.

2.6.3 Practical BFT: Cryptography Applied

Practical BFT (PBFT) [9] was one of the first algorithms to provide a protocol for consensus in an asynchronous and Byzantine environment and produced a practical implementation with significantly high throughput for message processing. PBFT is a state machine replication algorithm (discussed in §2.5.1) where a state is replicated on distributed systems that interact to reach consensus and replicate the agreed upon values.

In PBFT, each server is called a *replica*. Each replica moves through configurations called *views*. Views are numbered, and in each view there is a *primary* (or leader) replica, and the other replicas are each called a *backup*. The underlying mechanism for establishing consensus is essentially similar to Paxos (discussed in §2.6.1), and a high-level overview of the PBFT protocol is provided below:

1. A client c requests a service from the *primary* with value α.

2. The primary broadcasts the request to all *backups*.

3. All Replicas process requests then send replies to c.

4. Client c waits for $f+1$ replies from replicas with a result β, where f is the number of faulty components.

5. If $f+1$ replies contain $\beta = \alpha$, then c's request was chosen as the consensus result (initial value proposed was α). Otherwise, β is established as the majority consensus for the chosen value.

6. The final value is broadcasted to the network, and committed in the state machine.

With the aforementioned requirements, PBFT guarantees that nodes reach the same ordering of execution requests and ultimately reach consensus. Moreover, while PBFT assumes that an adversary can delay messages and control faulty nodes, it assumes that messages cannot be delayed indefinitely [9].

The major feature introduced in PBFT is the addition of cryptography to the same foundation used in Paxos [40] and Chandra-Toueg [10] algorithms. PBFT incorporated cryptography into its algorithm to provide message authentication by using signatures and hash digests as well as using Message Authentication Codes (MACs) as an alternative for costly digital signatures [8]. Nodes share session keys with replicas to enable cryptographic computations; there is an assumption that keys have already been exchanged between nodes in the network.

We discuss below how PBFT satisfies cryptography properties identified in §2.6.2. Note the following elements do not represent the full PBFT protocol, but rather they show how cryptography is applied to the underlying consensus protocol (Paxos). Moreover, the process of verifying signatures and their sequence satisfies properties §2.6.2(a, b):

1. For a request message, a client c sends a signed request to a replica to execute a command ϵ in the following format (this step modifies Paxos' **PROPOSE** step in §2.6.1):

$$<\text{REQUEST}, \epsilon, t, c>\sigma_c$$

   where σ_c denotes a signature by entity c, and t is a timestamp that can represent the client's local physical clock value. The timestamp is used to ensure the property stated in §2.6.2(c, d) with regards to redundant data and freshness identifiers.

2. Another example is the node's message to prepare, accept, and broadcast values, similar to Paxos but with cryptography applied. We show a representation of a commit request, sent from replica i, that follows the same notation as the previous example (this step modifies Paxos' **COMMIT** step in §2.6.1):

$$<\text{COMMIT}, v, n, D(m), i>\sigma_i$$

   where v is the view (or round) number that the node is working within, $D(m)$ is the digest of client's message m, and i refers to the replica identifier. The digest $D(m)$ satisfies property §2.6.2(d) to include redundant data for validation. The identifier i is used to satisfy §2.6.2(c, d) since it acts as redundant data to identify a specific replica with freshness proof to prevent replay attacks.

2.7 Performance Analysis

We discuss in this section the performance of the algorithms examined in the previous sections, focusing on resilience, message complexity (communication costs), and the cost of cryptographic operations.

2.7.1 Resilience

Distributed consensus resilience is defined in terms of its ability to withstand a number of faulty (possibly Byzantine) components, denoted as f, as a fraction of the total number of nodes in the system, denoted as n, for the protocol to maintain its liveness and safety properties [6, 15, 16].

Paxos withstands $2f + 1$ faulty (non-Byzantine) processes [40, 41, 46]. Therefore, to maintain correct system behaviour and consensus, the fraction of non-faulty nodes in Paxos is $\frac{2}{3}$ of n (where n is the total number of nodes). In PBFT, there are three thresholds that must be maintained for consensus: the prepare phase requires $2f$ votes, commit phase requires $2f + 1$, and the client requires $f + 1$ replies to confirm values. Therefore, the resilience of PBFT considers the three aforementioned level requirements regarding f, and the resilience is given as $n > 3f + 1$. This result means that more than $\frac{3}{4}$ of n must be non-faulty and non-Byzantine for the algorithm to maintain correct system behaviour and consensus [15,20,45]. This is a more stringent requirement than Paxos' $\frac{2}{3}$ of n requirement, and adds to the overall cost of implementing consensus in a Byzantine system.

2.7.2 Message Complexity (Communication Costs)

For asynchronous systems, time is defined in terms of message occurrences, and the delays between messages that cause their ordering to create a timeline of events [39]. These delays define message steps—or *asynchronous steps*—that processes use to define their local logical clock [16]. Paxos needs two asynchronous steps from the time the client issues a command until a consensus decision is made (two phases) [16]. This is an optimal result for non-Byzantine consensus asynchronous steps. However, reaching Byzantine consensus requires at least $f + 1$ rounds of message exchange, while the number of nodes required to provide resilience is $3f + 1$ (as defined in the previous section). The message complexity (number of messages to be sent, i.e. communication cost) for an optimal Byzantine consensus, which applies to the described algorithms in previous sections, achieves consensus with $O(n \times f)$ messages [21].

Furthermore, PBFT and an implementation based on PBFT called Zyzzyva [37], both employ batching to reduce the overhead required to process cryptographic operations. Batching is the process of running simultaneous instances of the protocol, where the primary replica assigns tasks to backup replicas.

2.7.3 Cost of Cryptography

The major bottleneck, and the main hindrance for applying BFT algorithms in distributed systems, is the overhead caused by cryptographic operations. Moreover, system recovery can be a costly process if the recovering node needs to reissue and disseminate its newly created keys [29]. However, symmetric cryptography, in the form of MACs, is shown to be more efficient. Using MACs, a system can achieve up to twice the performance of using digital signatures [8].

Replacing public-key cryptography with MACs can lead to optimal throughput and minimise system overhead (throughput is the number of processed requests per second). Authenticators are much smaller in size, when compared with RSA-1024 digital signatures. However, this advantage of authenticators over RSA, as explained in [8] requires $n \leq 13$ (where n is the number of servers).

We aim through our implementation to reach a trade-off between usability and system performance, with regards to using Elliptic Curve Cryptography (ECC) as well as symmetric stream ciphers, to produce fast and reliable results, and decrease the cost of cryptographic operations (explained in details in *Appendix A3*).

2.8 Summary

We discussed in this chapter the main alternative financial system, Bitcoin, that shares our research goals. We explored Bitcoin's problems, with regards to requiring that the majority of computational power is owned by honest nodes. This requirement can be violated by a single powerful entity that controls more than half the computational power in Bitcoin. For that reason, we turned to the literature regarding distributed systems, to investigate feasible alternative consensus algorithms that can produce cooperative consensus. The concepts and algorithms outlined in this chapter define and influence our system design; we use Byzantine Fault Tolerance to provide cooperative communication between servers to reach global consensus. In the following chapters, we introduce the theoretical design of our system, focusing on BFT operations and the monetary transactions required for our distributed payment system.

3 System Model

In this chapter, we discuss the system's environment and assumptions. We provide information regarding the communication environment, failure model, cryptography uses and assumptions, as well as assumptions regarding adversarial capabilities.

3.1 Communication Environment

We work in an asynchronous environment where messages can be delayed, dropped or received out of order. We adopt the asynchronous model since we cannot set any time bounds on message delivery; otherwise an adversary with inside knowledge will manipulate message delivery delay times to be close to the bounds. This model and assumptions are practical for a distributed message-passing model that works over the Internet. However, like many Byzantine Fault Tolerance (BFT) algorithms, we adopt a weak synchrony model to ensure liveness (to circumvent the FLP impossibility result). In this model, used in [9], let t be the time when a message is sent, and $delay(t)$ is the time between sending the transaction and the time a transaction was received at its destination. The client continues to retransmit the transaction until it is received at its destination; we assume that $delay(t)$ does not grow indefinitely faster than t. We also assume reliable links between each server, and each server is connected to every other server over the Internet. Given that reliable links will eventually deliver messages to their destinations, weak synchrony provides liveness in an unpredictable Internet environment.

3.2 Failure Model

We adopt a Byzantine failure model, where nodes can behave arbitrarily: Send malformed messages, withhold transactions, stop processing transactions, send conflicting messages, etc. We also note that we put no restriction on clients' behaviour, their numbers or their failure model. This lack of client restriction opens the system for attack vectors (related to Sybil attacks [19,22]) that we must address (we will discuss these issues in the *Threat Model* section). Moreover, we implement a BFT model inspired by PBFT as discussed in [9] for executing consensus. As mentioned in the *Literature Review* section, PBFT implements consensus with the existence of $3f+1$

servers (where f is the number of Byzantine servers), which is the same resilience provided in our system (we use the same thresholds implemented in PBFT).

3.3 Cryptography

We use cryptography to validate messages and implement accountability. Every node signs its messages and we assume that cryptographic techniques are effective; adversaries cannot forge signatures (unless private keys are compromised), create hash digest collisions (i.e. produce the same hash digest from different data), or regenerate the data given only a hash digest. Given these assumptions, nodes cannot change the content of messages sent by other nodes without being detected. This prevention of message forgery is critical for our system architecture and the BFT algorithm implemented, since we rely on the integrity of messages that can be relayed by other nodes on behalf of clients.

Moreover, we rely on cryptography to store servers' results as provable votes that create a voting mechanism, which are used as proofs to justify actions and verify the authenticity of a currency. Therefore, if cryptography is implemented correctly, and is effective in producing accountability and integrity, then we can limit the damage caused by Byzantine nodes.

3.4 Adversary Model

We assume a non-static or adaptive adversary: An adversary can compromise a certain number of servers or clients, and compromise servers dynamically. Even during a dynamic compromise of servers, while the servers are involved in consensus operations, as long as the required threshold for non-faulty servers are maintained, an adversary cannot compromise the integrity of consensus.

Furthermore, we assume that the adversary does not control every link between the servers. Although the adversary can attempt to delay messages between clients and servers, we assume that this delay cannot be executed indefinitely and that nodes will recover the path to deliver messages. Therefore, we assume an adversary can break links between some servers, but not all of them. We also assume that the adversary cannot circumvent cryptography as outlined in the previous subsection: An adversary can choose to duplicate messages, stop relaying them, or send conflicting messages, but cannot forge messages. Moreover, an adversary without a majority number of servers cannot compromise the system; the system can recover from malicious behaviour, and prevent error propagation into non-faulty servers (this is the essence of Byzantine Fault Tolerance).

4 Threat Model

We discuss in this chapter the various attack vectors introduced by our system architecture. We consider Denial of Service (DoS) attacks, transaction double-spending and coin ownership, and privacy issues. This section is concerned with discussing the underlying problems raised by the aforementioned issues; we shall discuss possible solutions and limitations in the *Evaluation* section of this research.

4.1 Denial of Service

The main threat to our system is a Denial of Service (DoS) attack. Instead of implementing a Bitcoin model with decentralised nodes[5], the targets of a DoS attack in our system are the

[5]It is important to note that an attack on the large mining pools is similar to an attack on a distributed system that we discuss in this section.

distributed servers that generate consensus. DoS attacks can be generated internally, whereby a Byzantine or compromised server can send an excessive number of messages to other servers. Externally, an adversary may aim to generate an excessive number of messages, possibly with valid form and signatures, to exhaust servers' ability to process requests. The external scenario constitutes what is known as a Sybil attack [19, 22], which is a by-product of allowing a large number of clients who can enter and leave the system without any control over membership. This uncontrolled membership limits the systems' ability to identify nodes effectively, and control which nodes can generate requests. An adversary can create a large number of nodes in the network in order to generate excessive requests and deplete system resources [22], thereby rendering the system unusable through DoS attacks. The controlled membership of servers, whereby servers are selected prior to executing the protocol (and their public keys are included in the protocol software), ensures that Sybil nodes cannot participate in the consensus decision, yet we must ensure that the damage caused by Sybil nodes through DoS attacks is curtailed (we implemented DoS prevention mechanisms, discussed in the *Evaluation* section).

4.2 Double-spending and Coin Ownership

Double-spending is the act of transmitting conflicting transactions with the aim of deceiving multiple recipients to accept the same coin as payment. Therefore, when Alice attempts to send multiple transactions to transfer the same coin to a different entity, one of the transactions must be processed, and it need not be the first transaction sent to the system. The rest of the transactions must be discarded if they try to allocate the same coin to a different entity.

Moreover, if the system undergoes isolation, whereby servers can communicate only with a subset of the servers, while believing that other servers are unreachable (effectively creating undesirable consensus quorums), then the system faces a problem. This threat is similar to Bitcoin's 51% attack, where a fork in the network can create an alternative consensus regarding the global ledger. Quorums will produce *different* consensus results amongst them, and will drive the system into consensus schisms. Therefore, our system must be able to effectively prevent this problem and prohibit the validity of consensus if none of the quorums constitutes the required majority for consensus. We discuss in *Protocol* and *Implementation* sections how the implemented BFT protocol provides properties to defeat schisms and maintain majority consensus (by requiring thresholds of server votes, and consensus cannot progress unless the correct thresholds are maintained).

4.3 Privacy

Although privacy had not been set as one of the major goals in the initial stages of the research, we aim to provide as much privacy as possible while considering system performance and usability. We aim to avoid Bitcoin's public ledger model, where the transaction history of clients is embedded into the publicly available blockchain, and anyone can see others' transaction history (which can reveal real identities by applying data mining techniques).

We must also consider the issue of *fault-tolerance privacy* [9]. A Byzantine server can leak information regarding transactions and the money supply. Moreover, an adversary who compromises a server may be able to view all the information that passes through the system during the consensus protocol: An adversary can enter passive eavesdropping mode by maintaining a server's correct functionality and monitor the generation of coins and the flow of currency.

5 System Design

In this chapter, we provide an overview and discussion of the system design. The overview provides a justification and blueprint of core functionalities, which will be discussed in details in the *Protocol* sections. The designs are largely based on Laurie's blueprint [47, 48] as well as the PBFT algorithm [8, 9].

5.1 State of the Coin

We first consider the issue of storing the coin. We follow an idea presented in [47] to provide an acceptable level of privacy and anonymity. The design relies on creating different public/private key pairs for every transaction (or to be more precise, every transaction on a coin). The public keys are used to create a user's address, which are used to spend coins and receive payments. Unless IP addresses are tracked, transactions will not provide information regarding account balances of individuals[6].

The motivation for creating new public keys is to break the relationship between transactions. At the same time, a transaction involving Alice paying Bob is similar to a transaction issued by Alice to move coins to other addresses she owns: It is non-trivial for the servers or any adversary who views transactions (in case of a compromise of a server) to determine whether Alice moved her own funds, or she paid another entity. This mechanism in itself provides a reasonable—albeit imperfect—level of privacy especially in the case where a server is compromised or ceased to view its local data.

This form of privacy is somewhat similar to Bitcoin's pseudonymous use of addresses to provide privacy. However, it is important to note that we implement a few changes in the design to further break the link between transactions as follows:

1. We do not allow a currency to be broken down to lesser denominations.

2. We do not create transactions that have multiple inputs, that are a long chain of transactions' history of who paid who, and (possibly) multiple outputs (with the outputs being change the payer returns to herself). Rather, each coin is treated as a separate entity.

Creating new public/private key pairs for each transaction may create substantial overhead: Generating many key pairs raises usability and resource issues when it comes to maintaining keys as well as storage requirements. Furthermore, creating new pairs for every single coin in the transaction is problematic, since there is a problem of communicating a large amount of new addresses to the payer sent by the payee. Maintaining a large number of keys (although stored until coins are spent) may be impractical if coins are not spent for an extended period of time. Therefore, we discuss in the *Protocol* and *Implementation* sections a modification to batch multiple coins into a transaction, to enhance usability and decrease transaction overhead.

5.2 Coin Supply

We now consider the issue of creating the coin supply. At this point, it is important to briefly review Bitcoin's approach to this issue. Bitcoin's protocol sets a limit on how many coins can be mined (21 million), and the timeframe when Bitcoins are mined (every 10 minutes a reward of 25 Bitcoins is rewarded to a miner). These rewards are halved to limit the coin supply so

[6] A trivial solution to IP tracking would be using anonymity tools, such as Tor, which can hide the user's IP. A nontrivial solution would be hiding transactions' sources through broadcasting messages to other users' in the network, who will contact the servers on behalf of the sending user. With added cryptography (encryption), the latter approach can simulate Tor's onion routing design, but will not be considered for this research.

that no more coins can be mined after the year 2140, and the incentive to maintain the system and perform transaction verifications shifts to transaction fees. We shall design our system to reflect the same properties produced by the Bitcoin protocol. We adopt a slight modification to number the coins sequentially (to produce coin serial numbers as *coinIDs*) as described in [47].

A significant issue related to the coin supply is who gets the coin reward. We adopt the *Commit-then-Reveal* protocol for distributing the coin rewards to servers randomly (initially mentioned in [48]). We discuss more details about implementing the *Commit-then-Reveal* protocol in the Minting sections.

5.3 System Ledger

As explained in [47], "a currency consists of a finite pool of tokens, each representing some amount of 'value'", and this is our adopted definition of a *coin*. As discussed in a previous section, a *coin* is assigned to a public key (*address*). The system ledger *coinTable* will include the sequential number of coins (*coinIDs*), and the public key (*address*) which owns each *coinID*. Furthermore, servers will maintain ephemeral mint proofs to agree on what coins must be minted.

6 Protocol

This chapter provides a discussion of the system architecture and the protocol used in the system. We provide details regarding the operations conducted by various entities in the system outside the BFT consensus operations, and discuss our modified implementation of BFT which is inspired by PBFT [9]. We discuss the notations used throughout the protocol and mention assumptions that are similar to those used in BFT algorithms.

6.1 Overview

The use of cryptography to implement consensus requires the exchange of public keys prior to running the consensus protocol. This requirement is emphasised in Laurie's blueprint [47] and BFT algorithms [8, 9, 14, 37]. Therefore, we assume a managed membership of the servers which involves the exchange of public keys prior to the execution of the protocol. We implement Elliptic Curve Cryptography (ECC), and Elliptic Curve Digital Signature Algorithm (ECDSA) as our signature method. Users' client programs will include the public keys of all replicas, to be pre-configured before delivering the source code. Moreover, with regards to the critical issue of exchanging keys between clients, we follow Bitcoin's convention: The exchange of the payee's *address* (which is generated from the hash of the payee's public key) is required and is sufficient to create transactions (we discuss this method in detail in later sections). The mechanism for exchanging the address is not included within the current protocol. We assume, similar to the Bitcoin model, that users will safely exchange addresses through another medium. At this point, we do not discuss how new servers can join and participate in the consensus protocol; we assume a static membership of servers.

Malicious clients' effect on the system must be limited. The Byzantine File System (BFS) implemented in [9] deploys an access control mechanism to prevent, or at least manage, clients' ability to write garbage data to the system. Similarly, since our system is based on permissions to change the ownership of a coin, clients' ability to change ownership of coins they do not own will be blocked, and must be monitored to flag abnormal behaviour.

For consistency, we follow most notations included in [9] with minimal changes. We denote a message m signed by node i as $\langle m \rangle_{\sigma_i}$. The public key of entity i is denoted as pub_c, and the hash of the public key which produces user x's address is represented as $address(x)$. The serial

number identifying a currency is denoted as $coinID$. The set of servers (each called a *replica*) is referred to as R, each replica is uniquely identified in the set { 0, ..., R - 1 }, and we assume f is the maximum number of faulty replicas. We show below a high-level overview of the protocol (the functions mentioned below are explained in subsequent sections), then proceed to provide more details regarding each component:

1. A client sends a transaction to any of the replicas, requesting a change of coin ownership (using **transaction()** function).
2. The receiving replica broadcasts the request to all other replicas (using **notify()** function).
3. Replicas execute the request, and prepare the result of the transaction for the client (using **change_ownership()** function).
4. The client waits for $2f + 1$ identical and signed replies from various replicas, relayed by any of the replicas, representing the result of the client's request (using **recv_reply()** and **transaction()** functions).

6.2 Coins

Each coin must contain essential information used in the protocol to verify transactions and perform ownership modification. The information stored in a freshly minted coin is as follows: $\langle coinID, address(x), t, V \rangle_{\sigma_V}$, where $coinID$ is the coin's serial number, $address(x)$ is the hash of the public key of entity x, t is a timestamp to record the time of minting, and V is the hash digest of the mint proof (see section *Minting* for more details). The coin is identified by its $coinID$ in the log, which is required for lookup and performing transactions on the coin. Entity x is the owner; x can spend the coin or change the owning address to another address x owns (provided x can prove ownership as explained in the following subsections).

6.3 Addresses

We use Bitcoin's method of creating user addresses; addresses declare the owner of a coin, as discussed in the previous subsection. We use ECC to create public/private keypairs. The *address* is represented in Base58 custom encoding[7] as the hash digest of a client's ECC public key, as well as a concatenated checksum value to verify the validity of the address. The main reason for using Base58 is to avoid visually ambiguous characters that may lead to mistyping an address (such as 'l' and 'I', or 'O' and '0'), as well as other practical issues such as avoiding breaks when sending values via email. We provide below an overview of Bitcoin's address creation process[8], that we adopt for our protocol (Figure 5 contains a diagram of the process[9])

[7] Created by Satoshi Nakamoto, C++ implementation available at: https://github.com/bitcoin/bitcoin/blob/master/src/base58.h

[8] Obtained from https://en.bitcoin.it/wiki/Technical_background_of_Bitcoin_addresses

[9] Obtained from https://en.bitcoin.it/w/images/en/9/9b/PubKeyToAddr.png

1. Create ECC key pairs (ECDSA will be used for signing messages using the client's private key).
2. Use the ECC public key created in the previous step to create the following (total will be 65 bytes):
   a. Prepend 0x04 (1 byte) to signify the address type.
   b. Extract X-coordinate (32 bytes) and append to the result above.
   c. Extract Y-coordinate (32 bytes) and append to the result above.
3. Hash the ECC public key using SHA256 hash digest function[10].
4. Hash the result generated in step 3 using RIPEMD160. Hashing twice is used in Bitcoin to avoid any vulnerabilities of using RIPEMD160 directly with ECDSA output. Bitcoin uses double hashing to generate an output using SHA256 then create a shorter output using RIPEMD160.
5. Prepend version byte to the result obtained above (this is to signify which type of network the address is used in; 0x00 is used to signify the main Bitcoin network).
6. Hash the result of step 5 using SHA256.
7. Hash the result of step 6 using SHA256 (double hashing is used as suggested by Ferguson and Schneier in [25] to avoid SHA256 length extension attacks).
8. The first four bytes of the result obtained in step 7 constitute the address checksum.
9. Append the address checksum obtained in step 8 to the result of RIPEMD160 digest obtained in step 4.
10. Convert the result obtained in step 9 into Base58, the result is the client's new address.

[10] SHA256 and RIPEMD160 are cryptographic hash functions which reduce file contents to 256 bits and 160 bits respectively (usually represented in hexadecimal characters), and are used to verify data integrity

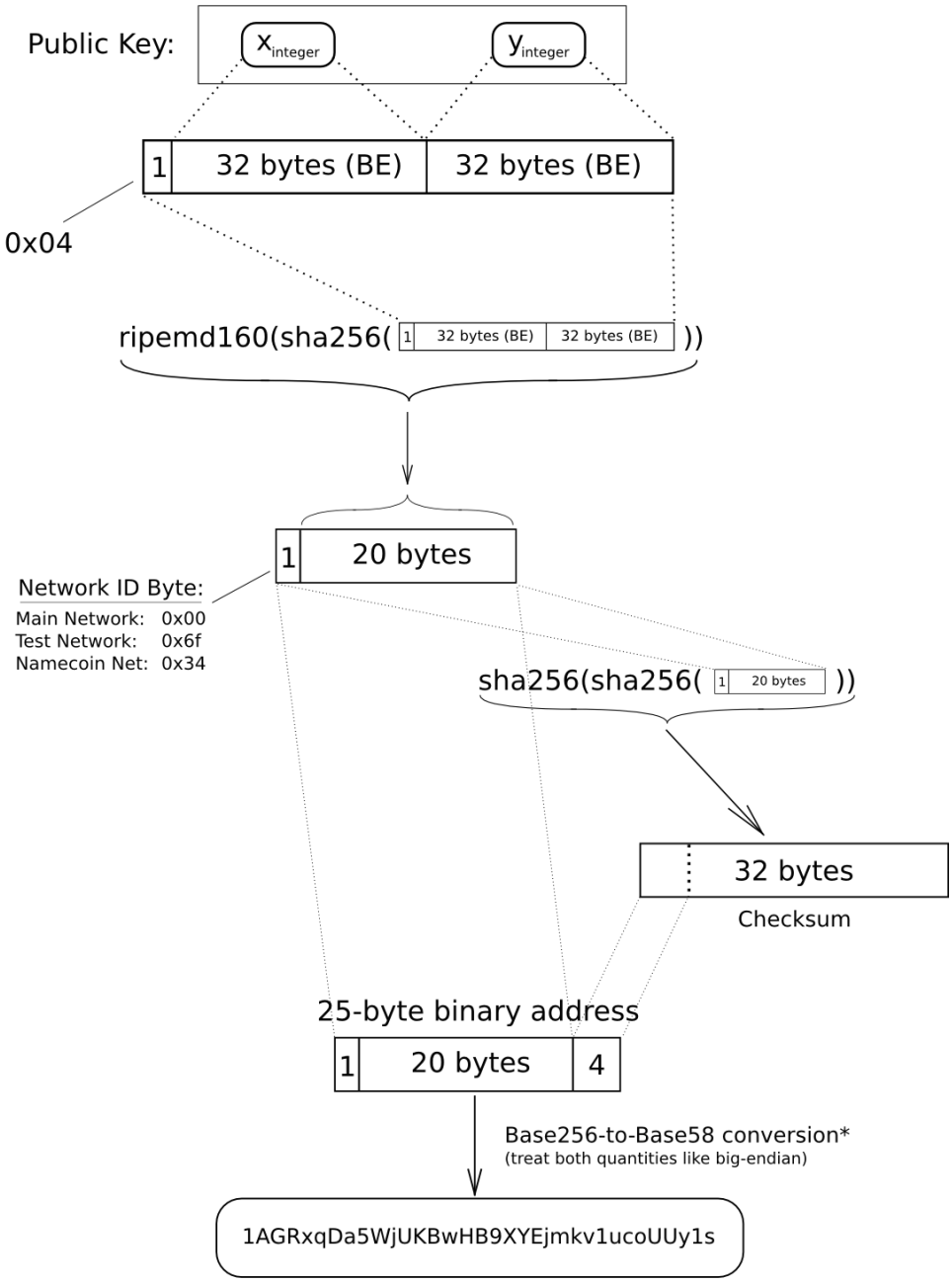

Figure 5: Bitcoin address creation process using ECC public key

6.4 Fundamental Operations

We outline below the fundamental operations used in the protocol. We discuss how to use cryptography to sign messages and verify signatures, as well as data structures to store and modify system information.

6.4.1 Digital Signatures

As explained earlier, we use public key cryptography to implement signature and verification schemes that provide integrity and accountability. We implement ECC and ECDSA to accomplish these tasks, and we follow conventional methods applied widely in cryptographic schemes to sign messages and verify signatures. The procedures outlined below are largely based on [58].

6.4.1.1 Creating Signatures The procedure for creating a signature takes as input the message to be signed m (which can represent any data, such as the *coin* abstraction identified in §6.2), and the signer's private key pri_i. The process includes creating a hash digest h of the data m, using SHA256 hash function. The signature algorithm then includes choosing the leftmost bits of the output as l, as well as selecting a random number k to be the multiplier of the generator G, such that $k \times G = (x, y)$. We calculate two results for the signature: $r = x \bmod n$ (where n is the order of the curve), and $s = \text{inverse}\ (k)\ (\ l + r \times pri\_i) \bmod n$. If either r or s are equal to 0, then the signature function must be called again. The result is the pair (r, s) to be sent along with the message m. The interface is shown in Algorithm 1.

def create_signature(m, pri_i):

input : m: message (or transaction)
pri_i : private key of entity i

output: (r, s) : message m signature

function: The purpose of this function is to produce an ECDSA signature for m, using pri_i. The resultant is the pair r and s, which depends on the hash digest h of m, the predefined ECC parameters, as well as the random point generation using a multiplier k. The pair (r, s) is sent along with m as the signature.

Algorithm 1: Creating Signatures

6.4.1.2 Verifying Signatures The receiver of the ECDSA signature for message m, which is identified above as (r, s), can verify the signature granted that the receiver has the signer's public key pub_i. The process of verifying a signature includes checking that the received results (r and s) are positive integers and are less than the curve order n. The receiver then calculates the hash digest h for m, and chooses l in a similar method discussed in the signature creation process. The receiver must calculate $w = inverse(s) \bmod n$, which is used to calculate $u1 = lw \bmod n$, and $u2 = rw \bmod n$. Then, the receiver must calculate the curve point $(x1, y1) = u1 \times G + u2 \times pub\_i$. Finally, the receiver checks whether $x1$ is equal to the received value r. If the values are equal, then the receiver accepts the signature and the function returns a Boolean value $True$. Otherwise, the receiver rejects and discards the signature, and the function returns $False$. The interface is shown in Algorithm 2.

def verify_signature(m, pub_i, (r, s)):

input : m : message
pub_i : entity i's public key
(r, s): message m signature

output: Boolean value $True$ or $False$

function: The purpose of this function is to verify an ECDSA signature for m, using pub_i and the signature pair (r, s). The resultant is a Boolean value $True$ or $False$, which indicates if the verification was successful or whether the signature does not verify given the input parameters, respectively.

Algorithm 2: Verifying Signatures

6.4.2 Database Operations

We implement hash table structures, whereby coin serial numbers are the keys used to implement database operations on a specific coin. Reasons for using hash tables include flexibility and performance; the average-case time for search, insert and delete is $O(1)$, and the worst-case time is $O(n)$ where n is the number of records in the hash table.

We include below simplified interfaces for the functions **htableSearch** , **htableInsert** and **htableDelete**, which perform lookup, insertion and deletion respectively. Note that the coin abstraction, which is inserted into the table, includes the coin details discussed in *Coins* section (§6.2).

6.4.2.1 Hash Table Search In order to search the hash table, the process includes checking whether the hash table entry is similar to the coin we are searching for. If the results are equal we return *item*, otherwise we continue to traverse the hash table. The interface is shown in Algorithm 3.

def htableSearch(*table*, *coin*):

input : *table* : hash table
coin : key to find

output: *item*: hash entry matching wanted key *coin*

function: The purpose of this function is to traverse the hash table in order to locate *coin*, returning the hash entry that matches the *coin* we are searching for (if a match is found).

Algorithm 3: Hash Table Search

6.4.2.2 Hash Table Insert Hash table insert operation is similar to hash table search. We check the hash table entries until we locate an empty slot. At that empty slot, we insert the *coin* value. The interface is shown in Algorithm 4.

6.4.2.3 Hash Table Delete Hash table delete follows the same steps outlined in the insert and search operations, with the only difference being that we remove the contents of the chosen element once located. The interface is shown in Algorithm 5.

def htableInsert(*table*, *coin*):

input : *table* : hash table
coin : key to insert

output: Hash table is modified, *coin* is inserted in empty slot.

function: The purpose of this function is to traverse the hash table in order to locate an empty slot, and to insert *coin* in that empty slot.

Algorithm 4: Hash Table Insert

def htableDelete(*table*, *coin*):

input : *table* : Hash table
coin : key to delete

output: Hash table is modified, *coin* is deleted if located.

function: The purpose of this function is to traverse the hash table in order to locate *coin*, and to delete the entry for *coin* from *table* if located.

Algorithm 5: Hash Table Delete

6.5 BFT Consensus Operations

We discuss in this section the BFT consensus operations which are a critical component of our system architecture. We include below a summary of the main operations to be used throughout the protocol to execute consensus operations (which are necessary to understand the various operations conducted by different nodes as explained in later subsections). We do not cover all functions available, but briefly include the main system calls used by various entities in the system.

6.5.1 BFT Approach

We have implemented BFT operations from scratch, inspired by PBFT, and modified to prioritise resilience over higher throughput, as well as provide the required authentication and financial operations for our system. We show in Figure 6 the modified operations of PBFT. The major difference to note in our BFT implementation is that we remove the role of the primary replica (or leader), which is used in BFT implementations to co-ordinate BFT operations, and instead use a broadcast model where a replica sends messages to all other replicas. Another difference is that we let the client query any of the replicas for the result of the transaction after a pre-defined wait time (rather than the replicas relaying the information directly to the client). More details about the differences between our approach and other BFT implementations are provided in the *Implementation* section.

6.5.2 Consensus Operations and Phases

When a replica receives a client's request, it initiates the consensus operations to reach agreement regarding the request. The three phases of BFT consensus are notify, prepare, and commit (we include a summary of PBFT operations that inspired our research in *Appendix A2*). Note that we changed the name of the pre-prepare phase to be the notify phase, to avoid confusion with prepare phase. At the beginning of each phase, a replica awaits a threshold of signed and valid replies from other replicas in order to progress the system through the phases, until a final

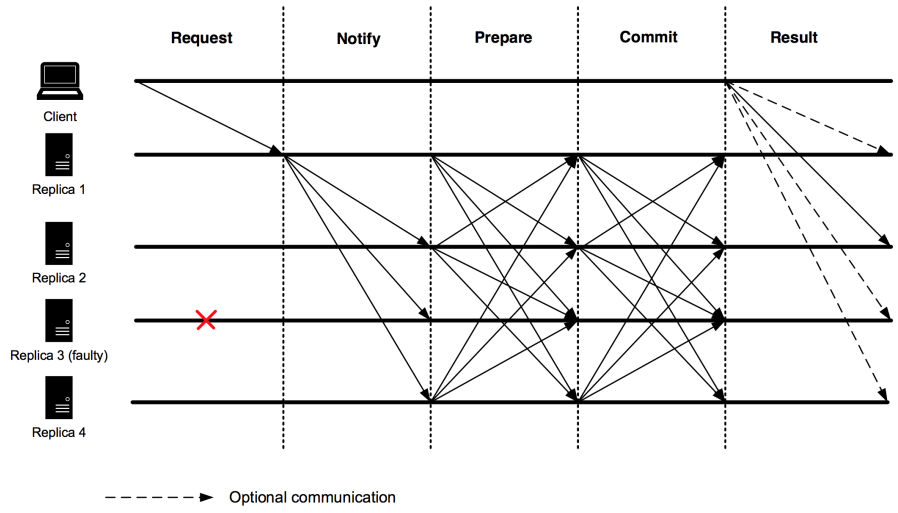

Figure 6: BFT consensus operations flow of messages

agreement is reached to enter into the commit phase. We separate the agreement segment of the consensus operation to include a prepare and commit phase. The prepare phase ensures that other replicas agree that a transaction is valid. The commit phase ensures that consensus was reached, and that non-faulty replicas agree to execute the consensus result (ensures a transaction will be executed on a majority number of non-faulty replicas). In our system, replicas agree to modify coin ownership if the transaction was successful or discard the transaction if it was unsuccessful. We include the PBFT consensus interfaces in the following subsections, which we modify to fit our system requirements as discussed in the *Implementation* sections.

6.5.2.1 Notify Phase A replica initiates the consensus procedure with the **notify()** function (to enter the notify phase). The input to the function is the requests queue $reqs\_queue$ which contains the requests (or transactions) to be executed. The output is a *notify* message, signed by the sending replica, and prepared to be sent to other replicas to initiate the consensus procedure. The interface is shown in Algorithm 6.

def notify($reqs\_queue$):

input : $reqs\_queue$: queue holds the requests to be executed

output: A notify message, signed by the sending replica

function: The purpose of this function is to create a signed *notify* message, which is signed by the replica that received the client's transaction, to be sent to other replicas to initiate the consensus protocol.

Algorithm 6: Notify Phase Message Creation

6.5.2.2 Prepare Phase Upon receiving enough valid votes to progress the system into the next phase, replicas enter the prepare phase by calling the **prepare()** function. The function is similar to the **notify()** function, but in this case replicas send the hash digest h of the request or transaction m (instead of the request itself, which had already been sent in the notify phase). The input to the function is the hash digest h. The output is a signed prepare message, to be sent to other replicas to progress the system into the commit phase. The interface is shown in Algorithm 7.

def prepare(h):
input : h: hash digest of request (the actual request was sent in the notify phase)
output: A *prepare* message, signed by the replica
function: The purpose of this function is to create a signed *prepare* message, to be sent as a vote to progress the system into the commit phase.

Algorithm 7: Prepare Phase Message Creation

6.5.2.3 Commit Phase Upon receiving enough valid votes from various replicas, replicas initiate the final phase of the consensus, the commit phase, in order for all replicas to execute the requests relayed in the notify phase. The input to the function is the hash digest h. The output is a signed *commit* message, to be sent to all other replicas to execute a transaction or request. The interface is shown in Algorithm 8.

def commit(h):
input : h: hash digest of request (the actual request was sent in the notify phase).
output: A *commit* message, signed by the sending replica.
function: The purpose of this function is to create a signed *commit* message to be used to end the final stage of consensus and execute requests (transactions).

Algorithm 8: Commit Phase Message Creation

6.6 Node Interaction

We outline in the following subsections how nodes interact in our system. We discuss fundamental operations such as sending and receiving requests, which cover the network operations in our system. In the following subsections, *node* refers to a replica or client.

6.6.1 Send Request

A client or replica prepares a message m to be sent to any other node. In our system design, a message can be a reply to a replica's message regarding one of the consensus phases, or it can be a transaction sent by a client. The contents of m for our architecture differ from PBFT, and we will define the contents in details in later subsections. For now, we mention that m is an array type that holds various data types identifying the message, which can include information related to verifying the coin ownership used to execute transactions, or replica's identifying information to be verified by other replicas. A node prepares the information m it wants to relay, and identifies which node should receive this information. The input of the function is m, as well as the

identification information *id* for which other node should receive the information. The output is a *Boolean* value which indicates whether the operation for transmitting requests was successful or not. The transmission is successful if the message was sent to its intended destination, a request is recorded in outgoing request array *out_req*, and a timer *rtimer* is started to measure the time it takes to process a request. A reply must be received related to m, within a time frame t, after which the request is considered expired and a retransmission of m is necessary. The interface is shown in Algorithm 9.

def send_data(m, *id*):

input : m: message or data to be sent
id: identifier of recipient of information

output: Boolean value *True* or *False* to indicate successful or unsuccessful transmission respectively. A timer *rtimer* is started when the message is sent.

function: The purpose of this function is to send m. The recipient is identified by *id* (which can refer to a pre-defined replica identification number or an IP address). This function inserts the transaction in the outstanding request array stack *out_req*, to monitor which transactions are being processed. A timer *rtimer* is initiated to monitor progress and ensure that the transaction is executed before time t passes.

Algorithm 9: Send Data Function

6.6.2 Receive Reply

A node awaits a reply through calling **recv_reply()**. There is no input to this function; the node listens for connections and receives relayed information. If the reply is valid and the receiving node successfully verifies the attached signature, then the received reply *rep* is the output of the function. The interface is shown in Algorithm 10.

def recv_reply():

input : None.

output: *rep*: received reply

function: The purpose of this function is to listen for incoming replies regarding previously sent transactions. The reply *rep* is validated by checking its signature, and is returned as the output of the function.

Algorithm 10: Receive Reply Function

6.7 Coin Operations

We now discuss the theoretical design for our payment system, based on the fundamental operations discussed in §6.4. We discuss how a client creates a transaction, how transactions are sent to replicas, and how replicas initiate the consensus procedure once a transaction is received.

6.7.1 Client Transactions

A client c issues a request to change the ownership of a coin: c signs a TRANSACTION message m, and sends m to any of the replicas in the following format: $\langle$TRANSACTION, $coinIDs$, $address(x)$, t, $pub\_c\rangle_{\sigma_c}$, where TRANSACTION identifies the type of message (we will use hexadecimal values to identify message types), $coinIDs$ contains a list of coins c wants to spend, $address(x)$ is the address of x who will be assigned as the new owner, t is a timestamp from the client's local clock, pub_c is c's public key, and m is signed by c as indicated by σ_c. This shows the requirement that public keys must be included in every transaction; public keys are used to verify signatures and coin ownership (which will be explained in the *Implementation* sections). Once m is prepared, m is sent to a randomly chosen replica. Note that c creates a signature $signM$ of m, and includes $signM$ in the data to be sent to replicas.

After sending a transaction, the client receives the result of the transaction using **recv_reply()** (as discussed in earlier subsections). Therefore, **recv_reply()** is called after a short wait period $waitTime$, if sending a transaction was successful. If a client does not receive a reply after some time, or if its transaction was rejected, it can send the request to another replica and extend $waitTime$.

Replicas will prepare their replies for when the client queries for a transaction result. A client requires $2f + 1$ identical results with valid signatures from replicas, relayed through a randomly chosen replica queried for the transaction result (received using **recv_reply()** after waiting for $waitTime$). These replies constitute the result of the transaction, and determine the consensus result, based on authenticated and valid replies from other replicas. Algorithm 11 shows the abbreviated pseudocode executed by the client to create a transaction, sign it, choose a random replica to send the transaction to, receive a reply and retransmit a transaction if necessary. Figure 7 shows the client transaction work flow.

def transaction:($coinIDs$, t, R, pub_c, pri_c, $address(x)$, $waitTime$, out_req):

input : $coinIDs$: selected coins
t : local timestamp
R: array of replica public keys and unique identification number
pub_c : client public key
pri_c : client private key
$address(x)$: address of new owner x
$waitTime$: time to wait for reply
out_req: array used to store hash of created transactions

output: None. **send_data()** will log the transaction in out_req and send the data to a chosen replica.

function: The purpose of this function is to enable a client c to create a new transaction message m, based on a list of *coinIDs* that c wants to spend, a timestamp obtained from the c's local clock t, as well as the public key pairs pub_c and pri_c. There are no outputs of the function; m is logged in the outstanding requests array out_req and **send_data()** is called to transmit the data to a randomly chosen replica. The client waits to receive replies, and retransmits m if $waitTime$ had passed.

```
// transaction message type set to value = 1
m ← [1, coin, address_x, t, pub_c];
signM ← create_signature(m, pri_c);
// log m into local repository of requests
out_req[0]= m;
// Generate random number i between 0 and R.length()
rand_i ← random number;
// concatenate m with signM, and send data to Replica_rand_i
send_data(m + signM, rand_i);
while rtimer < waitTime do
  recv_reply();
end
// Generate random number i between 0 and R.length()
rand_i ← random number;
send_data(m + signM, rand_i);
// Reset timer and wait for a longer time after retransmit
rtimer ← 0;
while rtimer < 2 × waitTime do
  recv_reply();
end
```

Algorithm 11: Client Transactions Procedure

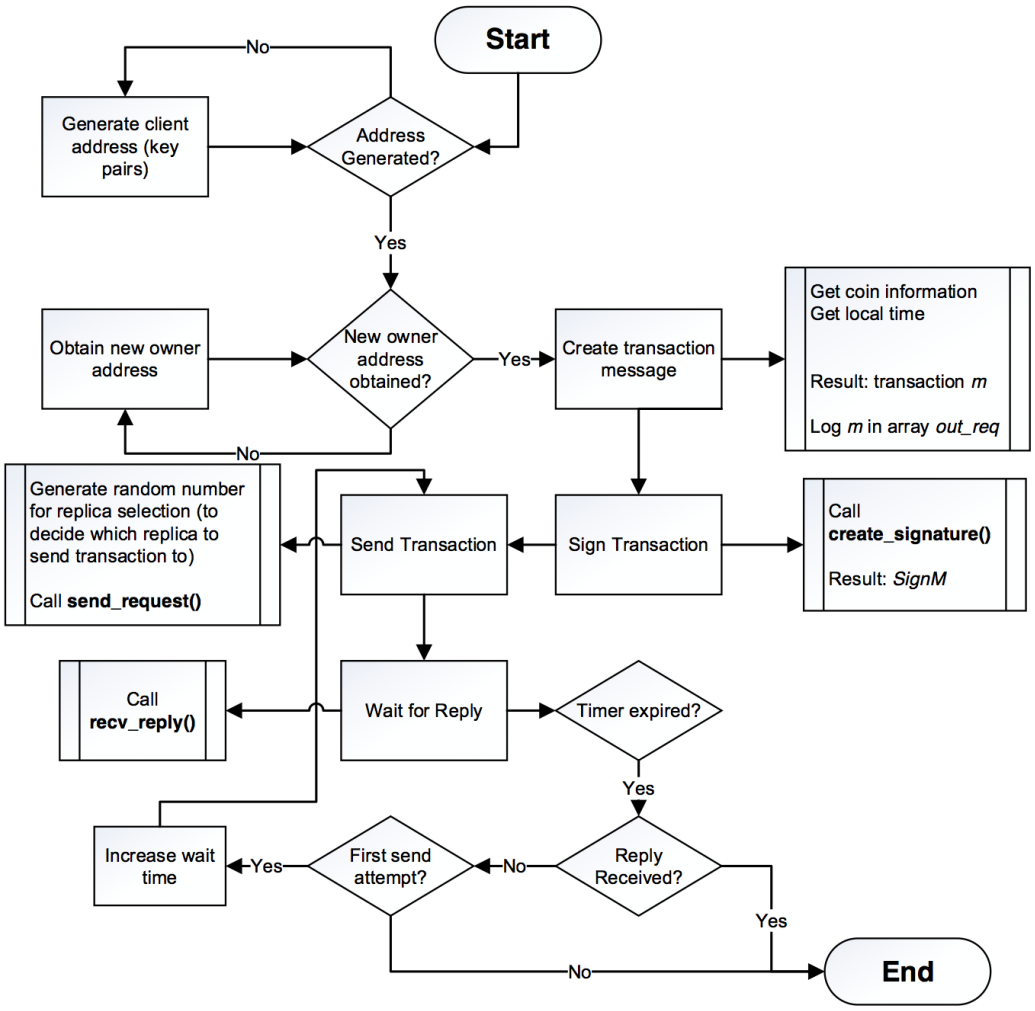

Figure 7: Client transaction work flow

6.7.2 Changing Coin Ownership

We now discuss the operations for assigning a new owner to a coin. Message m and its signature $signM$ are received directly from the client. A hash digest of m is created as h to check if the message is already in the queue array $Queue$, which stores transactions to be processed (if m is already in $Queue$ (identified by its hash h) and is awaiting processing, m is then discarded). If m is not in $Queue$, a replica then proceeds to process m, by first verifying $signM$ using **verify_signature()** which is explained in §6.4.1.2. If the signature is valid, the replica proceeds to verify the address using the attached pub_c. The hash digest of pub_c must generate the same $address\_current$ which is stored in $coin$ details in the coins hash table $coinTable$. If the address generated by pub_c and the stored $address\_current$ are different, then c is not the owner of the coin and the transaction is invalid. Otherwise, ownership has been validated, and the procedure

continues to assign new ownership of the coin. It is important to note that $address\_current$ is the locally stored value of the owner's address of the coin located in $coinTable$, while $address(x)$ is the address included in m which instructs replicas to assign ownership to x (i.e. c is the sender of the transaction and wants to pay x). Once the owner is verified, a replica initiates the BFT consensus protocol to assign ownership (by sending a *notify* message as discussed earlier). If the consensus is successful, the replica performs a hash table insert **htableInsert()** to replace the coin's $address\_current$ with $address(x)$. Algorithms 12 and 13 (two parts) show the abbreviated pseudocode executed by replicas to verify m and initiate consensus to assign a new coin owner and prepare transaction results (more details about the practical implementation of this process are included in the *Implementation* sections). Figure 8 shows the change ownership work flow.

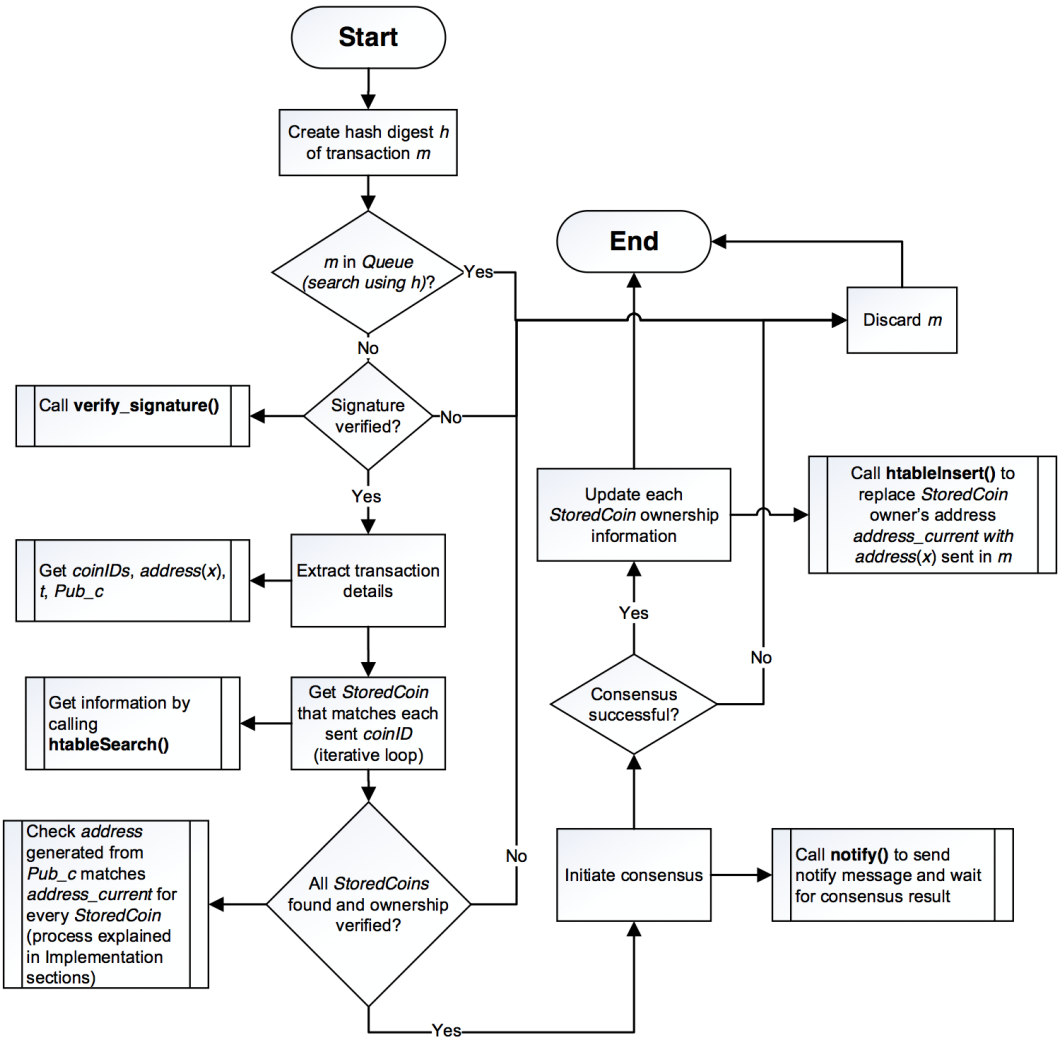

Figure 8: Change ownership work flow

def assign_owner:($m + signM$, $Queue$):

input : m : message, array containing [x01, $coinIDs$, $address(x)$, t, pub_c], where
x01 : identifies transaction message type (hexadecimal value),
$coinIDs$: list of coin identification numbers that client c wants to spend,
$address(x)$: the address of the new owner,
t : the local time stamp at the client,
pub_c : public key belonging to the client (payer)

$signM$: message signature
$Queue$: queue (array data type) of hash of messages being processed

$coinTable$: Local repository of coins and owners (not a function input, but is used in this function)

output: None. Local hash table $coinTable$ is modified to include $address(x)$ as new owner of $coinIDs$.

function: The purpose of this function is for a replica to validate and verify a client c's transaction m. First a replica checks if m had already been received, and if it is being processed the replica discards m. If the address is valid and the signature is verified, the replica checks the ownership of the *coinIDs* listed in m. If c owns the listed coins, then the replica initiates the consensus protocol using **notify()**. If the consensus is successful, a replica assigns $address(x)$ as the new owner of *coinIDs*.

```
// create hash of the message for tracking
h = hash(m);
// discard m if it is being processed, checking hashes in Queue
for i in Queue do
    if h = Queue[i] then
        discard m
        exit
    end
end
// extract information from m
coin = m[1];
address(x) = m[2];
t = m[3];
pub_c = m[4];
```

Algorithm 12: Assign Owner Procedure - Part 1

```
// verify signature
if verify_signature(m, pub_c, signM) = true and  address(c) is verified then
    // hash table lookup operation for coin
    StoredCoin = htableSearch(table, coin);
    if StoredCoin != ' ' then
        // modify the coinID owner address to be the
        // value sent by the previous owner in m
        StoredCoin[1] = address(x);
        // initiate BFT consensus operations
        // req_queue holds requests to be processed
        req_queue[0] = m;
        notify( req_queue[]);
        wait for consensus phases to complete;
        if consensus is successful then
            htableInsert(coinTable, StoredCoin);
        end
        else
            discard m;
            exit
        end
    end
    else
        discard m;
        exit
    end
end
else
    discard m;
    exit
end
```

Algorithm 13: Assign Owner Procedure - Part 2

6.8 Minting

We discuss in this subsection the minting process to choose a replica to receive a reward, which effectively controls the money supply in our system. We follow the idea presented in [48] to create coins and assign the newly created reward to a replica. Recall that replicas are numbered in the set R, and their numbering is preconfigured in the software implementation. Each numbering is associated with a public key of that particular replica and is also preconfigured. Therefore, each replica sends a mint commit message (example for $replica_i$): $\langle \mathsf{REWARD}, commit \rangle_{\sigma_i}$, where *commit* is the commitment value. The *commit* value is the hash digest of the replica's $mint\_vote$ (a randomly chosen integer) and a *nonce* (random data). A replica reveals its $mint\_vote$ and *nonce* once all commitments are received. Once enough mint votes and nonces are revealed (which are valid, signed and produce the corresponding replica's previously sent *commit*), a replica decides the recipient of the reward as follows: $recipientID = \sum_{i=0}^{R} (mint\_vote_i) \bmod \mathsf{R}$. The minting process is outlined in the pseudocode shown in Algorithms 14 and 15 (two parts). Figure 9 shows the minting work flow.

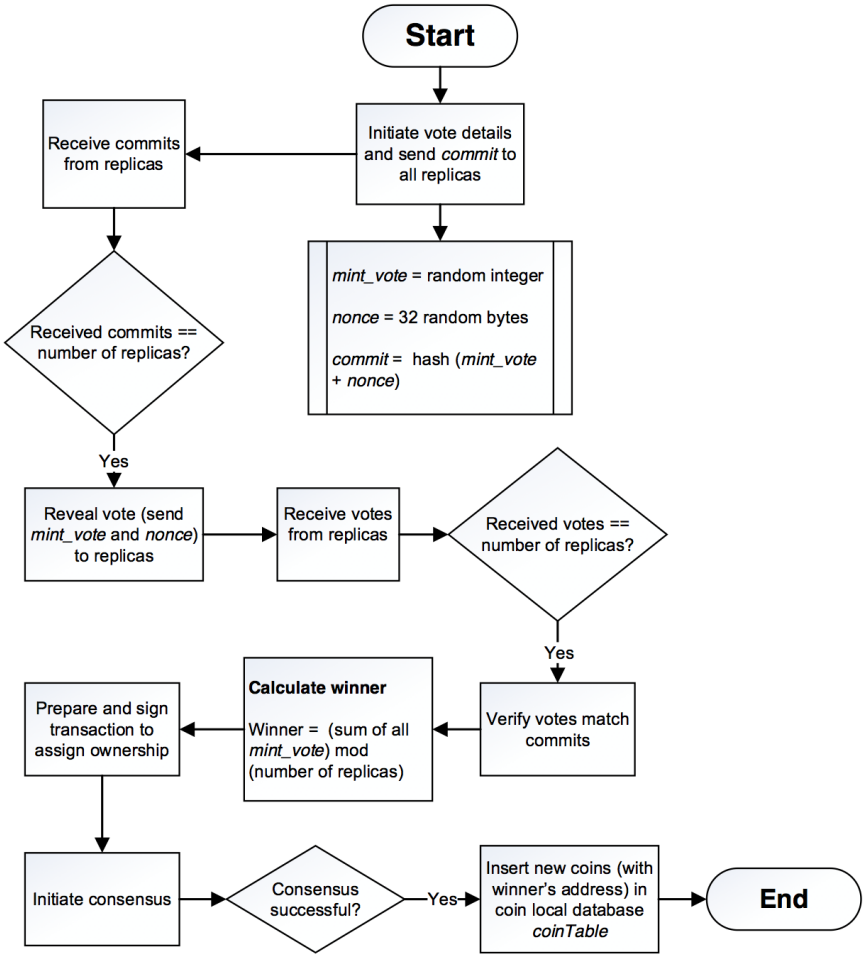

Figure 9: Minting work flow

Having calculated *recipientID* and identified the replica to be rewarded the mint coin rewards $coinIDs$, a $replica_i$ initiates the consensus protocol, to assign ownership to the new mint coins by signing and broadcasting a special transaction $m = \langle \mathsf{MINT}, coinIDs, address(recipientID), h(V)\rangle_{\sigma_i}$, where *coinIDs* is a list of numbers for the coins to be minted (starting at a sequence number larger by one than the currently highest *coinID* in $replica_i$'s local database), and $h(V)$ is the hash digest of the mint votes' stack V, sent as proof of the *commits* and $mint\_votes$ received from other replicas (we explain the mint proof V in the *Implementation* section). The system then proceeds to process this transaction, and will finally assign membership of the coin to the appropriate replica. All replicas will log the newly minted coin to their local coin database if the consensus protocol is successful.

def mint:(R, key_table, $coinID\_latest$):

input : R : array of replica public keys and unique identification number
key_table : hash table containing replica keys in local repository
$coinID\_Latest$: most recent $coinID$ serial number

$coinTable$: local repository of coins and owners (not a function input, but is used in this function)
pri_k : replica's private key (not a function input, but is used in this function)

output: None. Local hash table *coinTable* is modified to include newly minted coins.

function: The purpose of this function is for a replica to create its own reward vote, gather votes from other replicas, and calculate the recipient of the reward. Based on other replicas' votes, a replica calculates the winner of the reward and prepares a transaction to assign the reward coins to the winner. The replica signs the reward and broadcasts it to other replicas, and if consensus is successful (enough votes agree to assign ownership to the same winner), the local hash table for coins *coinTable* is modified to include the newly minted coins.

```
// reward_type is an integer value representing reward messages
reward = [reward_type, commit] ;
for i in R do
 | send(reward, i)
end
initialise commit[];
commit[my_replica_name] = mycommit;
// receive replica commits in commit[]
i = 0;
while i < R do
 | commit[i] = received_commit
end
if commit.length == R.length then
 | reveal my vote (send my mint_vote to replicas) and store values in V[];
end
```

Algorithm 14: Mint Procedure - Part 1

```
// calculate recipient
i=0;
sum=0;
for i in V do
|  sum = sum + V[i]
end
recipientID = sum mod R.length();
// get recipient_address from local repository
recipient_address = htableSearch(key_Table, recipientID);
// prepare transaction m with mint reward coinIDs
coinID = coinID_Latest + 1;
m = [coinID, address(recipientID), hash(V)];
signM = create_signature(m, pri_k);
Queue ← m;
// assign_owner(), is explained in Implementation sections
assign_owner(m + signM, Queue);
if consensus is successful then
|  htableInsert(coinTable, coinID);
end
```

Algorithm 15: Mint Procedure - Part 2

7 Implementation

In this chapter, we discuss the practical implementation of the protocol. We provide details about major components; we primarily focus on BFT operations, coin operations, and minting. We also discuss our BFT modifications, providing information regarding how and why we modified existing BFT protocols. We also provide in Table 1 the list of implementation components, and include details about their implementation in *Appendix A3* and *A6*. We have decided to encrypt every network transmissions in our system using ECC to provide better security through confidentiality (and we use Rabbit stream cipher for shared secrets key, refer to *Appendix A3* for details). Therefore, clients and replicas send encrypted transmissions, and other nodes are unable to decrypt the messages unless they have the correct private key.

Component	Purpose	Location
Test Environment	Information regarding the offline lab environment	A3-1
Cryptographic Operations	Explain Elliptic Curve Cryptography (ECC) and how it is used in our implementation, as well as the implementation of hash functions	A3-2
Generating Addresses	Information regarding addresses creation, and other local information	A3-3
Database Operations	Information regarding how Python `dictionary` data types are used to create and manipulate hash tables	A3-4
Network operations	Information regarding how Python's `socket` module is used to implement TCP streams	A3-5
Miscellaneous Components	Extra features implemented in the current version of the protocol (DoS prevention and encrypted email capabilities)	A3-6
User Documentation	Brief user documentation for starting either server (replica) or client operations; to introduce users to the software implementation (this information is included in the software source repository in `README.txt`)	A6

Table1 - List of implementation components

7.1 BFT Operations

We implemented BFT operations from scratch and did not use any external library. This is the core component of our implementation, found in `byzops.py` in the implementation source code, and includes consensus phases, voting mechanism, minting and coin operations, and many critical hash tables (Python `dictionary` data types). In the following subsections, we discuss our consensus approach, the implementation of BFT operations and consensus phases, as well as vote processing. We postpone the discussion of coin operations to a dedicated section.

7.1.1 Consensus Approach

After considering PBFT [9] and Zyzzyva [37] implementations of BFT operations, we chose not to follow the same approach used by those implementations. Both implementations use a designated primary (replica leader) to handle sequencing of transactions. A performance bottleneck is introduced by the primary: a faulty or overwhelmed primary can degrade overall system progress substantially. In fact, during our initial implementation, we implemented a version of PBFT using a primary replica, and realised that *view changes* (required to replace a faulty or overwhelmed primary, mentioned in *Appendix A2*) led to an almost complete halt of system performance and the rejection of new transactions (clients have to wait until a system stabilises itself again). Moreover, a primary is used to issue sequences to transactions and handle commit results (in the case of PBFT). We overcome both requirements in our system design, as we explain in later subsections. Furthermore, the discussion above is also supported by [13, 43], where the authors criticise PBFT and Zyzzyva for the same reasons, and indicate that system progress is critically affected by a Byzantine leader.

Therefore, we modified our implementation to reflect a major divergence from our initial consideration. We do not designate any replica as a primary, and we implement a broadcast model to send every message to all replicas. The design that closely reflects our implementation is briefly discussed in [43], where a leaderless approach to Byzantine Paxos allows replicas to autonomously decide what steps must be taken next, without requiring a leader to coordinate actions. We discuss below our reasoning for creating a different approach to eliminate the need for a primary replica.

7.1.1.1 Sequencing We do not require sequencing as defined in PBFT and Zyzzyva. In those implementations, sequencing is required to ensure that transactions are not executed out of order. Since PBFT authors used their implementation to create Byzantine File System (BFS), the researchers aimed to prevent transactions from being committed out of order (writing data out-of-order results in garbage data that must be reordered). However, in our system, commitments are governed by ownership of a coin. Replicas will process transactions, check ownership, and initiate consensus regardless of which transaction is received first. Take for example two transactions that depend on each other, i.e. Alice pays Bob with coin x, then Bob pays Charlie with the same coin x: Bob can never pay Charlie with x unless Alice's transaction was committed first and Bob was assigned as the owner of x (we will revisit this assessment in the *Evaluation* section).

7.1.1.2 Information Relay Commitment of a transaction is governed by replica votes regarding a transaction, and a commitment is executed once thresholds are met. We do not relay information directly to the client. Instead, we designed clients to wait for a short period of time before requesting results of transactions. This allows replicas enough time to process a transaction, prepare messages regarding a transaction (if it is successful) and store the message in hash tables *Payer* and *Payee*, designated to relay results to clients. A client requests transactions' results from any replica, and processes the sent proofs to validate that transactions were successful.

7.1.1.3 Garbage Collection This process is coordinated by a primary in PBFT, and is required to ensure identical information on every replica. Moreover, this process is used to agree on sequences and system stages that, as we discussed earlier, are irrelevant to our implementation. We have designed a payment system depending on coin ownership stored in replicas' local coin databases (*coinTable*). However, we do not require identical information of *all* coins on *all* replicas; we require identical information regarding a *particular* coin on a *threshold* of non-faulty

replicas. For example, replicas 1, 2 and 3 can agree on the status of coin x, while replicas 2, 3, and 4 agree on the status of coin y: Given that the other replicas are non-faulty and proceed with the consensus phases, in the first case replica4 will be forced to change the ownership of x regardless of its vote. In the second case, replica1 will be forced to change its vote. By "forced", we mean that if a replica wants to maintain a correct view of the system, this replica will change ownership details based on the voting result, since the result reflects the system's consensus. In fact, a faulty replica can enter *passive-mode* where it does not send votes but executes consensus results. This faulty replica will be able to change ownership information to reflect non-faulty replica threshold votes, and will be able to add coin details to its coin repository *coinTable* over time. We have implemented the system in this manner to speed up transaction processing, and to allow a faulty replica to recover from total disk failure (operating without its coin repository *coinTable*). We tested the implementation to allow a replica to fully reconstruct its *coinTable* based on others' consensus results. Therefore, a specially designated process for garbage collection, that severely degrades system performance and halts progress, is unnecessary for our implementation and is overcome by utilising BFT consensus itself.

7.1.1.4 Broadcast Model Using a broadcast model increases network resource usage and message complexity (the number of sent messages increases significantly), yet this increased usage is practical considering the benefits obtained by eliminating the primary bottleneck. Instead of sending messages to the primary which coordinates actions, replicas send messages to every other replica. Each replica then decides what the required next step is for a transaction, and sends the next consensus phase message to all other replicas, or a replica halts processing a transaction if the required threshold of votes has not been received. We will discuss the communication cost in the *Evaluation* section.

7.1.2 Consensus Phases

We implemented three consensus phases: Notify, prepare, and commit. We limit our discussion below to the notify phase; the other two phases are similar in their processing of transactions (each phase sends a different message type, `x02`, `x03`, `x04` respectively). We also provide abbreviated code for the notify phase (which is representative of other phases' operations). Prior to the notify phase, a replica checks coin ownership based on the information relayed by a client. A message is rejected if checking ownership fails. However, a replica may have incorrect ownership information in its *coinTable*, and a client may indeed have sent a correct transaction. Therefore, a client can send the same transaction to another replica if a transaction fails. Our decision to reject a transaction if checking ownership fails is motivated by the need to minimise spam and prevent Denial of Service attacks (see *Evaluation* section). We provide the notify interface and abbreviated source code in Algorithm 16.

The notify phase is the only phase where a transaction m is included in the messages sent to other replicas. All subsequent phases use the hash digest h of m instead. The **notify()** function is called by the replica that received m, to initiate consensus and send m to the other replicas in a notify message. The recipient replicas call **notify()** as well, to record information regarding m, but those replicas do not send any information to other replicas, since the notify message was already relayed by a replica (local data is modified and m is logged in this case). The *Messages* hash table is used to store transaction information and replica messages, and h is the key for *Messages* entries. We designate the notify message type value as `x02`.

The other two phases, prepare and commit, send similar information sent in the notify phase. The difference is that each phase progresses only when enough votes are gathered, and the initialisation placeholders for signatures are replaced with actual signatures to be broadcasted.

In the case of the prepare and commit phase, each replica broadcasts data, using the **broadcast()** function (similar operations to **send_data()**, but the data is sent to all replicas), to every other replica (unlike the notify phase, where only the recipient replica sends m to other replicas). Another significant difference is that in the prepare phase, replicas send *ownership_vote*; a replica sends a Boolean value indicating that the sender indeed owns the coin. We discuss the vote processing mechanism and thresholds in the next subsection.

7.1.3 Vote Processing

Another critical component in `byzops.py` is the **add_vote()** function. This function is called when a replica receives any transaction type message (`x00`, `x01`, etc.). We pass a *type* integer to **add_vote()** to signal the processing for a specific message: Prepare votes are *type 2*, commit votes are *type 3*, and so on. The function then retrieves the current vote stack (hash table), using the transaction hash h as key for the *Messages* hash table. The vote stack uses a replica's name as the table key (to record its votes and data), and its message regarding a particular transaction for a particular phase. Then, **add_vote()** updates the vote stack to store the verified phase messages, and counts the votes for that particular transaction in that phase (replicas count their own votes as well). For the prepare phase, the threshold required is that the variable *vote_yes* (votes indicating the sender is the owner) is equal to or greater than $2f$. For the commit phase, the threshold requirement is set to $2f+1$ which means that a majority of replicas must agree that the client owns the coins, and have sent their promise to execute the transaction (in a commit message). We show in Algorithm 17 the interface and simplified source code for **add_vote()** function. The different threshold requirements reflect that we designed the system to progress when more than double the number of faulty nodes f vote to progress the system (speed up processing), while we require the majority number of replicas to send their promise to execute a transaction to ensure that consensus is global. This means that with 4 replicas deployed in the test environment, we require that 3 replicas are online and non-faulty for the final commit to be executed. Note that since the system resilience (discussed in *Literature Review* section) is given as $n > 3f + 1$ (where n is the total number of replicas, and f is the number of faulty replicas), then $f = \frac{n-1}{3}$. Moreover, we have designed the system to be asynchronous as we discussed in the *System Model* section; replicas can proceed to process other transactions while they wait for the required votes. In other words, transactions are executed in parallel, independently from any other transaction. A replica's vote may be delayed, so it will be added to the stack as soon as it is received by other replicas.

It is important at this stage to discuss the locking mechanism implemented in our design. Although we considered the locking mechanism provided by Python's `threading` module (which seemed to cause a halt until it finishes processing *Messages* in memory), we decided to implement a simplified version based on Boolean variables used as flags to signal the end of a phase. For example, if enough votes are gathered for a commit phase, a non-faulty replica executes the transaction (changes ownership) and sets the $set\_commited\_flag$ variable to *True* (or 1). Votes may still be received after a threshold has been reached. Therefore, If a replica later receives more commit votes, and it has already changed $set\_commited\_flag$ to *True*, it adds the later votes to its stack as proof of validity, but does not proceed to execute the transaction again. This simple locking mechanism provides a local signal to the replica, to either add a vote or proceed to execute ownership changes. Note that this is possible since we load one version of *Messages* into memory; it would be an error to load *Messages* into memory for every connection thread, because it would cause different and contradicting versions of the coin ledger to exist in a replica's memory.

def notify:(m, $signM$, h):

input : m: transaction message
$signM$: client c's signature of m
h: hash digest of m

pri_k : replica's private key (not a function input, but is used in this function)

output: None. Consensus is initiated using **notify()**, and m is logged locally.

function: The purpose of this function is create local information regarding m (storing contents of m, using its hash digest h as a key for the entry into the hash table *Messages*). This information will be required to process m and advance through consensus phases. If a replica received m from a client, then that replica broadcasts m to other replicas. Otherwise, if a replica learned about m from another replica, it only logs m locally.

```
1  sign_hash = create_signature(h, pri_k)
2
3  #create notify message type \x02 (as well as hash and signature)
4  notifymessage = ["\x02", sign_hash]
5  hash_of_notifymessage = hashthis(notifymessage)
6  notifymessage_signature = create_signature(hash_of_notifymessage, pri_k)
7
8  #initialise prepare (\x03) and commit (\0x4) placeholders
9  #for the transaction, note: socket.gethostname() is used
10 #to get hostname value
11 preparevotes = {}
12 preparemessage = ["\x03", 0, 0, h, socket.gethostname(), 2]
13 hash_of_preparemessage = hashthis(preparemessage)
14
15 #no need for actual signature, we are only initialising at the moment
16 preparemessage_signature = 0
17 preparevotes[socket.gethostname()]=
18   (preparemessage,preparemessage_signature)
19
20 #set_prepared_flag is a placeholder, changed when 2f prepare
21 #votes are gathered, the status "prepared" means enough votes
22 #are gathered and commit should be initiated
23 set_prepared_flag = 0
24
25 commitmessage = ["\x04", 0, short_id(h), h, socket.gethostname(), 2]
26 hash_of_commitmessage = hashthis(commitmessage)
27
28 #no need for actual signature, we're only initialising at the moment
29 commitmessage_signature = 0
30 commitvotes = {}
31 commitvotes[socket.gethostname()]=
32   (commitmessage, commitmessage_signature)
33
34 #set_commited_flag is also a placeholder, see comment for
35 #set_prepared_flag above
36 set_committed_flag = 0
37 Messages[h] =
38   (m,signM, h, notifymessage, notifymessage_signature,
39   preparevotes, set_prepared_flag, commitvotes, set_committed_flag)
40
41 #pseudocode is shown below
42 if receiver of m:
43   broadcast(data)
```

Algorithm 16: Notify Function Implementation

def add_vote:(*type*,*m*, *signM*):

input : *type* : type of **add_vote()** code to process (2 for prepare, 3 for commit, etc.)
m: message from replica (prepare, commit, etc.)
signM: replica signature of *m*

output: *Boolean* value for status of adding votes.

function: The purpose of this function is to process replica votes, and modify vote stacks (hash tables). The function adds votes and checks if thresholds are met, and progress the replica to send the next phase message (in prepare type), execute ownership change (in commit phase), or call **assign_reward()** to assign mint reward coins after a successful mint vote process (discussed in later sections).

```
1  #note that only prepare phase is shown in the code below;
2  #commit and mint have similar processing (different thresholds for f)
3
4  #prepare message are type 2
5  if type == 2:
6     #load data for that transaction (using hash h as key) from memory
7     m,sign,h .... modify_preparevotes ... =  Messages[h]
8
9     vote_no = 0
10    vote_yes = 0
11    for i in modify_preparevotes:
12       if modify_preparevotes[i] == 0:
13          vote_no = vote_no+1
14       elif modify_preparevotes[i] == 1:
15          vote_yes = vote_yes+1
16
17    if vote_yes >= ((2*f)):
18       return 1
```

Algorithm 17: Vote Processing Implementation

7.2 Coin Operations

We now discuss the mechanism implemented for creating transactions, as well as checking and changing ownership of coins. This section relies on database operations (discussed in *Appendix A3*). Coin operations are included in `byzops.py` (which is the same file that contains the consensus operations discussed in §7.1).

7.2.1 Transactions

To create a client transaction we allow the user to provide input regarding the receiver of the funds and the amount to be sent, and we pass those as variables ($address\_x$ and $numberOfCoins$ respectively) to the transaction function, along with the users' key pairs and *clientCoinTable* (which is the client's coin database, stored on disk as `clientcoin.data`, and is loaded at startup). The terminal output displayed in Listing 1 shows the process of getting user input as a result of running `client.py` and choosing the "Send Payment" option. If a client confirms the input, **transaction()** is called to generate a transaction message to be sent to a randomly chosen replica. The resultant data will be sent to a randomly chosen replica by calling **send_data()**. We provide the interface and simplified code in Algorithm 18.

```
[05/25/14 11:19:21] client [INFO] configuring payer mode
[05/25/14 11:19:21] client [TRAN] You have 150000 coins

Enter the address you want to send coins to
(Press Enter to accept default as 12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL)
>12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL

[05/25/14 11:19:24] client [TRAN] setting coin recipient as
12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL

Enter number of coins you want to send to 12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL
(Press Enter to accept default as 700)
>45
[05/25/14 11:19:29] client [TRAN] setting number of coins as 45
--------------------------------------------------
Confirm your transaction
--------------------------------------------------

[Recipient] 12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL
[Amount] 45
[Your Address] 12mr5KkjEeyjE4BTJMZ7XVuGXUmkprgobVuq

Is the information above correct? y/n
(You can press Enter to accept default as 'y' )
> y

---> Information confirmed, proceeding with transaction
```

Listing 1: Creating a transaction using the client program

def transaction:(*pub_c*, *pri_c*, *numberOfCoins*, *address_x*, *clientCoinTable*):

input : *pub_c* : client *c*'s public key)
pri_c: client *c*'s private key
signM: replica signature of *m*
numberOfCoins: number of coins to spend
address_x: recipient of coins (the new owner if the transaction is successful)
clientCoinTable: client *c*'s coin database, stored on disk as `clientcoin.data`

output: *m*: transaction
signM: client signature of *m*

function: The purpose of this function is to generate a transaction message *m* to be sent to a random replica. The function loads *coinIDs* from the local database *clientCoinTable* (stored on disk as `clientcoin.data`), to create a list of coins to spend *coinList*, which will be inserted into the contents of *m*. Transaction *m* includes client *c*'s public key *pub_c* (required for checking ownership) as well as a timestamp using Python's `time` module. The transaction is signed as *signM* using c's private key *pri_c*, and the function returns *m* and *signM*, which constitute the data to be sent using **send_data()**.

```
1  coinList=[]
2  if ( pub_c == '' or pri_c == ''):
3    return 0, 0
4  else:
5    counter=0
6    while counter < numberOfCoins:
7    try:
8      for i in clientCoinTable:
9        coinID = i
10       if coinID not in coinlist:
11       coinList.append(coinID)
12       counter=counter+1
13       break
14   except:
15     pass
16
17 if (coinList != ''):
18    m = ["\x01", coinList, address_x, time.time(), pub_c]
19    signM = create_signature(hashthis(str(m), pri_c)
20    return m, signM
21
22 else:
23   return 0,0
```

Algorithm 18: Transaction Implementation

7.2.2 Check Ownership (Authentication)

When a replica receives a transaction m from a client, it checks coins' ownership by calling **check_ownership()**. The same function is called when replicas are determining the status of coin ownership after they received a notify message from a replica which initiated the consensus process. Ownership checks depend on two critical verifications:

i. Verifying the signature using the public key *pub_c* attached to m (*pub_c* is located in the 5th slot of m (i.e. $m[4]$)).

ii. Generating the address of the attached public key as *received_address*, and checking whether *received_address* is equal to the current owner's address *address_current* for each coin listed in m.

Coin information is retrieved from *coinTable* using the *coinID* as key for *coinTable*. If verification (i) is successful, it ensures that the corresponding private key was used to sign the message (checking authenticity of the message). If verification (ii) is successful, it ensures that the current owner of the coin matches the sending client's details (*pub_c*'s hash digest matches the current owner's address information and, thus, this verification validates ownership of the coin). The two verifications provide the authentication mechanism required for ownership changes in our system: They ensure the sending client owns the coins, and signals the replica to return a value indicating a valid transaction. If any coin is not owned by the client, then the transaction is rejected as invalid. We show in Algorithm 19 the simplified code for **check_ownership()**.

def check_ownership:(m, *coinTable*):

input : m: client c's transaction
coinTable: replica coin database (the local ledger) stored on disk as `coin.data`

output: *Boolean* indicating whether a check on all coins was successful

function: The purpose of this function is to generate *received_address* using client c's public key *pub_c* included in m, and check if *received_address* is equivalent to the current owner's address *address_current* for every *coinID* sent in m. If all checks for every coin is successful, we return 1 indicating a successful transaction. .

```
address_x = m[2]
pub_c = m[4]
received_address = gen_address(pub_c)

for i in coinIDs:
  sent_coinID = i
  if coinTable.has_key(sent_coinID):
    coin = coinTable[sent_coinID]
    address_current = coin[1]
    if address_current == received_address:
      valid_transaction = 1

  else:
    return 0

if valid_transaction:
   return 1
```

Algorithm 19: Check Ownership Implementation

7.2.3 Change Ownership

The change ownership function, **change_ownership()**, is called after replicas agree to commit a transaction (i.e. consensus is reached). The committed transactions are stored in another hash table (*ExecutedTransactions*) and messages are created in *Payer* and *Payee*, to relay messages regarding transactions to clients and payees respectively. For every *coinID* included in *coinList* in m, a replica retrieves the corresponding coin and its stored information using *coinID* as key for *coinTable*. If a coin is found, the stored owner address field, which is *coin*[1], is modified to be $address\_x$ which was sent in m. We show in Algorithm 20 the simplified version of the code.

def change_ownership:(m, *coinTable*):

input : m: client c's transaction
coinTable: replica coin database (the local ledger) stored on disk as `coin.data`

output: *Boolean* indicating whether a check on all coins was successful

function: The purpose of this function is to retrieve the information of the stored *coin*, for each coin included in m, using *coinID* to query the *coinTable* hash table. The contents of the coin's current owner is replaced with $address\_x$ which was sent in m.

```
1  coinList = m[1]
2  for i in coinList:
3     sent_coinID = i
4     new_owner_address_x = m[2]
5     if coinTable.has_key(sent_coinID):
6        coin = coinTable[sent_coinID]
7        i = coin[0]
8
9        #replacing contents of coin with new information (new_owner_address)
10       coinTable[sent_coinID] = [i, new_owner_address_x]
11
12    #we provide here a mechanism to add and change ownership,
13    #based on consensus results. change_ownership will only be called
14    #when a replica received valid commit votes (that satisfy the threshold)
15    #for commit consensus (see vote processing section), this means that by
16    #reaching this step, coin status must be changed regardless
17    #of the local replica's vote regarding ownership
18
19    else:
20       i = m[1][0]
21       coinTable[sent_coinID] = [i, new_owner_address_x]
22
23 return new_owner_address_x
```

Algorithm 20: Change Ownership Implementation

We show in Listing 2 an example log output of the else statement in Algorithm 20, which is executed for each coin not found locally (when a replica does not have the coins locally, but is forced to add them to its *coinTable* based on verified commits, as explained in §7.1.1.3):

```
[05/25/14 13:29:21] replica4 [BYZ] did not find coins in my local coinTable. Since I
received valid commits that satisfy the threshold requirement, I will add coinID:
300096 to my coinTable based on replica consensus.
```

Listing 2: Log entry for inserting new coins based on verifiable consensus

7.3 Minting

The minting implementation is used to generate rewards for replicas and increase the coin supply. As discussed in earlier sections, we adopt the idea suggested in [47, 48], and discuss below the *Commit-then-Reveal* minting protocol, and the functions involved in the minting process.

7.3.1 Mint Time

Minting occurs after a pre-defined amount of time passes since the previous mint stage. We set the time between mint stages to be 10 minutes (600 seconds) to match Bitcoin's settings with regards to increasing the coin supply. We define a sequence variable *seq* to refer to the number of coins already minted, and *seq* will be used to define the new *coinIDs* (incrementing the maximum *coinID* from the previous mint). The *seq* variable should not be confused with the variable used for the ordered number of received transactions used in many BFT implementations. The process of checking *seq* and starting the minting process is handled in **mint_time()**. We include the simplified code in Algorithm 21.

7.3.2 Minting Process

When **mint_time()** returns a non-zero value, indicating a replica must start minting, the replica starts the minting process via the **mint_commit()** function. This process involves generating a random value *mint_vote* (between 1 and 10), which is then used to generate a replica's commitment value. The *commit* value is generated by producing a hash digest of the value *mint_vote* along with 32 bytes of random data (*nonce*), to be sent to all other replicas. A replica creates a message type `x00`, containing the value "*mc*" to identify a mint commit message, along with the name of the replica (obtained using **socket.gethostname()**), as well as the values *commit* and *seq*. Once the mint commit message type is prepared, the replica either obtains the current *mint_commit* stack from *MintCommit* table (a replica stores *commits* even if it did not reach that *seq* yet), or it generates a new *mint_commit* hash table if no such table exists for that particular *seq*. After either retrieving or creating *mint_commits*, the replica includes its *mint_vote*, *nonce*, and commit in *mint_commits*, then sends its commit to other replicas through calling **broadcast()**. We include in Algorithm 22 the simplified code.

Commit votes are added using **add_vote()** in a similar procedure used to process BFT consensus votes (as shown in §7.1.3). Once enough commit votes are received (replicas wait for commitments from all other replicas), a replica reveals its minting vote by sending its vote *mint_vote* and the value of *nonce* in a message type `x00` with a message value set to "*mr*" to identify a mint reveal message. Replicas then check that the hash digest of a replica's *mint_vote* concatenated with the received *nonce* match a replica's *commit*, in which case the replica's *mint_vote* is valid. If the commit and hash digest of received values do not match, this means a replica attempted to change its vote in the mint reveal segment of minting. This change in vote may indicate that a replica is attempting to change the outcome of the voting process to win the mint reward. For that reason, we chose to implement the *Commit-then-Reveal* protocol discussed above; replicas first commit to a value, then we reveal the votes. If a *commit* does not match the received values in the second round (mint reveal round) then a replica's *mint_vote* is discarded. We chose to omit the code for the **minting()** function, since it is similar to the **mint_commit()** function discussed above, with minor changes regarding message identifier and values sent (**minting()** sends values *mint_vote* and *nonce*). Replicas then proceed to assign the mint reward as shown in the next subsection.

def mint_time:(m, $coinTable$):

input : seq: maximum integer of $CoinIDs$ which have been minted in the previous mint stage (value is 0 if minting never occurred)

output: $status$: $Boolean$ 1 to signal minting, otherwise it is 0 to signal non-minting.

function: The purpose of this function is to determine whether enough time has passed since the last mint stage (600 seconds). If the necessary time between mint stages has passed, the current time is stored in the $mint\_timestamp$ (to be checked for subsequent calls) and the function returns a $Boolean$ value 1 to initiate minting. Otherwise, the function returns 0, indicating that minting should not occur since the required mint time ($threshold\_to\_mint$) has not been reached yet.

```
#Number indicating the length of time to wait (in seconds) between mint stages
threshold_to_mint= 600

#get current time
time_now = time.time()
if seq == 0: #genesis mint

   #must remove the following block in future production versions,
   #since checking seq==0 every time will slow the check process.
   #But we need this segment for continued testing
   #using an empty coin database (coin.data)

   #genesis_minted is a global variable
   if genesis_minted !=1:
      genesis_minted =1
      mint_time_stamp = time_now
      return 1

   else:
      return 0

elif time_now - mint_time_stamp >= threshold_to_mint:
   mint_time_stamp = time_now
   return 1

else:
   return 0
```

Algorithm 21: Minting Implementation (decide on mint time)

def mint_commit:(m, *coinTable*):

input : *seq* : maximum integer of *CoinIDs* which have been minted in the previous mint stage (value is 0 if minting never occurred)

output: None: Mint message x00 is sent to other replicas

function: The purpose of this function is to generate a random value *mint_vote*, and generate the replica's *commit* value by producing a hash digest of the value *mint_vote* along with 32 random bytes of data (*nonce*). A message is created containing x00 as the message type and value "*mc*" as the identifier of a mint commit message. The message also contains *seq*. The hash table *mint_commit* is then retrieved from *MintCommit* or created, to be modified with the replica's values for *mint_vote*, *nonce*, and *commit*. The replica then proceeds to broadcast its commit using **broadcast()**.

```
1  mint_vote = random.randint(1, 10)
2  nonce= str(os.urandom(32).encode('hex'))
3  mycommit = hashthis(nonce+str(mint_vote))
4
5  message = ["\x00", "mc", socket.gethostname(), mycommit, seq]
6
7  #pri_k is a global variable available for all functions
8  message_signature = create_signature(hashthis(str(message)), pri_k)
9
10 try:
11    mint_commits= MintCommit[seq][0]
12 except:
13    mint_commits = {}
14
15 mint_commits[socket.gethostname()] = (socket.gethostname(), mycommit, seq)
16 MintCommit[seq] = (mint_commits, 0, mint_vote, mycommit, nonce)
17
18 data =(message, message_signature)
19 broadcast(data)
```

Algorithm 22: Minting Implementation (initiate *Commit-then-Reveal* protocol)

7.3.3 Assigning Rewards

When enough verifiable mint votes are processed via **add_vote()**, replicas create a transaction and notify other replicas about the decision by calling the **assign_reward()** function. In this case, unlike regular non-mint transactions, all replicas broadcast their **notify()** function result (whereas for non-mint transactions, only the replica that received the transaction directly from the client relays a notify message). The first notify message received will be processed, and all subsequent notify messages referring to the same transaction will be discarded. The minting message, which is a special transaction to mint new coins, has the message type set to `x05`. This mint message, will be processed as a normal transaction, and consensus will be initiated to commit the transaction based on local replica verifications (as discussed in BFT operations §6.5). In this function, *seq* is used to determine the coinIDs of the coins to be minted, since replicas will only reach this point if agreement regarding *seq* is achieved in previous mint stages. Therefore, replicas will produce the same new *coinID* used to determine the *coinID* of all subsequent coins which will be newly minted. We add the hash digests of the mint message to $CoinsToBeMinted$ table to be verified in **check_ownership()** (modification explained shortly). We provide the interface for **assign_reward()** and the simplified code in Algorithm 23.

When a replica reaches the **check_ownership()** function (as it progresses through BFT stages) and does not find the coins in its local *coinTable*, it checks *CoinsToBeMinted*, and verifies that indeed it has the same hash of the message transaction in its local database (the hash digest of the message produced in **assign_reward()** as shown above). Therefore, it adds the coins to its *coinTable* based on the result of checking *CoinsToBeMinted*. We did not include the process of checking *CoinsToBeMinted* in the previous discussion of **check_ownership()** (in §7.2.2) since it would have seemed out of place without giving the required context. We provide the full short code for **check_coinstobeminted()** in Algorithm 24 as well as the block that processes *CoinsToBeMinted* in **check_ownership()** in Algorithm 25 (modification of **check_ownership()** is shown, the remaining simplified code was included in Algorithm 19).

def assign_reward:(*mint_result*, *seq*, *coinTable*):

input : *mint_result*: result of processing mint votes, obtained from **add_vote()**
seq: maximum integer of *CoinIDs*, minted in the previous mint stage
coinTable: replica coin database (the local ledger) stored on disk as `coin.data`

output: None. Special transaction message `x05` is sent to other replicas.

function: The purpose of this function is to assign the mint reward to the appropriate replica as a result of the minting votes (processed after the *Commit-then-Reveal* protocol discussed earlier). Replicas form a special mint transaction with type set to `x05`, which will initiate BFT consensus operations (by first calling **notify()** as explained in earlier sections).

```
1  #lookup_* functions are created to process
2  #replica information based on name, ip, etc.
3  replicaname = lookup_name(mint_result)
4  newCoinIDs=[]
5
6  newCoinID=seq+1
7  replica_address = lookup_address_from_id(mint_result)
8
9  while i < mint_reward:
10    newCoinIDs.append(newCoinID)
11    newCoinID=newCoinID+1
12    i=i+1
13
14 #create message \x05
15 #MintedSeq is a table containing mint proofs
16 m = ["\x05", newCoinIDs, replica_address, MintedSeq[seq], seq]
17 hash_of_m = str(hashthis(m))
18
19 #the following hash table is explained in a subsequent code listing
20 CoinsToBeMinted[hash_of_m] = m
21
22 signM = create_signature(hash_of_m, pri_k)
23 data = (m, signM)
24 broadcast(data)
```

Algorithm 23: Assign Reward Implementation

def check_coinstobeminted:(*hash*, *seq*, *CoinsToBeMinted*):

```
#hash refers to the hash digest of the mint transaction that the
#replica stored after reaching assign_reward

if CoinsToBeMinted.has_key(hash):
   return 1
else:
   return 0
```

Algorithm 24: Checking Mint Coins Status

def check_coinstobeminted:(*hash*,*seq*, *CoinsToBeMinted*):

```
coinIDs = m[1]
must_mint = check_coinstobeminted(hashthis(m), CoinsToBeMinted)
if must_mint ==1:
   address_x = m[2]
   for i in coinIDs:
      sent_coinID = i
      coinTable[sent_coinID] = [sent_coinID, address_x]
   valid_transaction= 1
```

Algorithm 25: Check Ownership Implementation (modification to check mint coins)

We show in Listing 3 an example of a minting record from a replica's log (note that the payer is recorded as "MINT" to indicate minting rewards):

```
[Transaction Reference] a7a62d_da
[Details]
Hash: a7a62d632a2b9cd525439b03bcd78f20d4251f4987b723358496cc66c41058da
Message text: MINT paid 12mn1b7fkznTNBLDrwXKbmaZrcMdU6RePD6T with 100 coin(s) in
transaction a7a62d_da
Date: 2014-05-25 13:29:21.974062
Payer: MINT
Amount: 100
Payee: 12mn1b7fkznTNBLDrwXKbmaZrcMdU6RePD6T
```

Listing 3: Replica's mint log entry

8 Evaluation

In this chapter, we evaluate the protocol after deploying 4 replicas on NeCTAR cloud (refer to *Appendix A5* for details about the cloud implementation). We discuss the running time for various critical components in the system (including transactions and minting), memory and disk usage, and communication and power consumption costs. Finally, we compare our results to Bitcoin's recorded peak performance (which occurred in January 2014) by minting the same amount of coins circulating in the Bitcoin network (12878450 coins at the time of writing).

8.1 Running Time

To compute time duration of various components in the system, we chose not to use Python's built-in modules `timeit` or `time.clock()` which measures CPU clock. The former calculates timestamps well for loops while the latter may produce inaccurate results since CPU times vary on different hardware. Rather, we chose to timestamp various components before and after completion using `time` timestamps, to give more accurate results of the various protocol components' running time. We discuss in the following subsections transaction and mint running times. We also provide a discussion about network propagation delays (and how it is effected by link speeds and number of replicas) after discussing message complexity.

8.1.1 Transaction Running Time

Transaction running time analysis is based on a sample of 4000 transactions. We define the running time for a transaction to include the total time required for the replica to perform the following steps (along with various hash table operations that are omitted for brevity):

1. Process client IP for spam and rapid transmissions.
2. Verify signature (using the public key included in transaction message).
3. Store client's public key in the *PublicKey* table.
4. Store transaction in *Messages* table.
5. Verify ownership.
6. Initiate consensus.
7. Complete the three phases of consensus (as defined in §6.5).
8. Change ownership of coins if consensus is successful (end of commit phase).

We timestamp a transaction when it is stored in *Messages* table (step 3 shown above) and subtract the timestamp generated in step 8 from step 3's timestamp to produce the total transaction running time. The running time, for a sample of 4000 transactions, ranged between 0.1508 and 0.3696 seconds (for each transaction). These results show that the *minimum* transaction times in the online cloud simulation match the *maximum* transaction times obtained while testing the implementation in a local offline lab, using a 1 Gbps LAN connection between servers (replicas). This provided preliminary indicators that the running time for transactions is closely tied to message complexity (cost of communication), as well as the network bandwidth for links between replicas, which affect the propagation times required to transmit messages (we will revisit this analysis in a later subsection).

The average transaction time (averaged over 4000 transactions) was 0.3365 seconds. This reflects the ability of the system to process an average of 2.97 (or almost 3) transactions per second. The following log entries show timestamps for a single transaction 302a52_9b (in both a replica's log shown in Listing 4, and the client's log shown in Listing 5):

```
[06/11/14 17:21:54] replica1 [TRN|TIME] timestamp for logging transaction 302a52_9b :
  2014-06-11 17:21:54.316462
[06/11/14 17:21:54] replica1 [TRN|TIME] it took 8.70227813721e-05 seconds to change
  coin(s) ownership status for transaction 302a52_9b
[06/11/14 17:21:54] replica1 [TRN|TIME] timestamp of completing transaction
  302a52_9b: 2014-06-11 17:21:54.664803
[06/11/14 17:21:54] replica1 [TRN|TIME] it took 0.348437786102 seconds to complete
  transaction 302a52_9b
```

Listing 4: Replica log for transaction 302a52_9b

```
[06/11/14 17:21:59] client1 [TRN|TIME] appended transaction 302a52_9b to transaction
  table (duration 1.59740447998e-05 seconds)
[06/11/14 17:21:59] client1 [CRYPT|TIME] Completed signature and encryption for
  transaction 302a52_9b in 0.00643181800842 seconds
[06/11/14 17:21:59] client1 [TIME] Completed sending transaction 302a52_9b in
  0.0309770107269 seconds. Waiting before asking for verification...
[06/11/14 17:22:02] client1 [TIME] Got result for transaction 302a52_9b in
  0.473391056061 seconds
[06/11/14 17:22:02] client1 [TRN|TIME|FIN] Completed transaction 302a52_9b at
  2014-06-11 17:22:02
[06/11/14 17:22:02] client1 [TRN|TIME|FIN] transaction 302a52_9b took 2.5066587925
  seconds to complete (including wait time before verification)
```

Listing 5: Client log for transaction 302a52_9b

8.1.2 Minting Running Time

Minting implements the *Commit-then-Reveal* protocol discussed in §7.3. This protocol requires that mint proofs regarding *commits* are resent to replicas for verification, and artificial delays (enforced in software) are put in place to delay the process while replicas verify and resend their votes. Therefore, minting requires longer times to complete than normal transactions. We believe that this delay is acceptable, given that the procedure requires more computing to be performed than a normal transaction (in fact, the minting process also includes creating and processing a transaction), and that replicas can wait to receive their reward while clients would demand fast transaction times. Minting running time for 500 mint stages averaged at almost 5 seconds. Listing 6 shows log entries and timestamps for the minting process.

```
[06/05/14 22:57:56] replica1 [MNT|FIN|TIME] minting for seq = 11913 ended at
  2014-06-05 22:57:56.641174
[06/05/14 22:57:56] replica1 [MNT|FIN|TIME] minting for seq = 11913 took
  6.45697999001 seconds to complete
```

Listing 6: Replica log showing mint transaction timestamps

8.2 Memory and Data Usage

By using Python's built-in module `pickle` to load data into hash tables, we are storing contents to be processed in memory (RAM). Loading 12 million coins to memory required 3084 MB of RAM. All other information stored in hash tables in memory are ephemeral and are not stored on disk. Therefore, we are mainly concerned with the implementation's memory performance with regards to handling the money supply (stored in `coin.data`). Loading a large number of coins puts a considerable demand on memory and requires a substantial amount of RAM, which limits the ability of running `sever.py` implementation on low-end computers such as Raspberry Pis or smartphones. Moreover, loading a substantial number of coins puts considerable demands on the CPU, when Python's module `pickle` loads coins as objects from disk. Loading about 12 millions, for example, takes 32 seconds, which is acceptable given that we load almost 1 GB worth of data into RAM (disk usage is explained shortly). Listing 7's log entries show that the replica stalls for about 32 seconds while loading coins.

```
[06/10/14 22:19:46] replica1 [INFO] loading coins...
[06/10/14 22:20:18] replica1 [INFO] loaded 12000050 coins in 32.0732150078 seconds
```

Listing 7: Replica log showing time required to load 12 million coins

This delay happens when restarting replicas, which should occur infrequently since the implementation does not require any restarts (but server downtime is normal for software or hardware updates, or due to full migration to other hosts). A replica that restarts is considered Byzantine while it is restarting (it is considered unavailable, unresponsive or withholding information) since other replicas will not be able to connect to it or receive its votes (Python's `socket` module fails after a timeout when attempting to create a TCP stream to an offline replica).

With regards to disk usage, a coin (including its various attributes) requires 54 bytes on disk. Therefore, generating 12878450 coins would require around 663.2197 MB on disk ($(54 \times 12878450) \div (1024 \times 1024)$). This is still acceptable when compared to Bitcoin's current numbers at 12 millions coins regarding disk usage (we will revisit the comparison in details in a later subsection).

8.3 Message Complexity (Communication Costs)

We calculate in this section the message complexity of various components in the system. Message complexity can be defined as the communication costs incurred in the network, i.e. the number of messages that must be sent to complete various components. We investigate both the minimum and maximum number of messages required in order to reflect the best and worst case scenarios, respectively, with regards to communication costs. It is important to note that since we changed our BFT implementation, our message complexity differs from previously mentioned analysis provided in the *Literature Review* chapter.

8.3.1 Transaction Message Complexity

From a client perspective, a transaction must be sent to at least one replica. Replicas that disagree with the ownership of the coins included in a transaction will reject it, and a client must resend the transaction to another replica. Therefore, the minimum requirement for sending a transaction from a client to a replica is 1 message, while the worst case requires sending the message to all replicas, i.e. n messages, where n is the number of replicas in the system.

8.3.2 BFT Operations Message Complexity

The complexity is more complicated on the replicas side due to BFT operations and various scenarios that must be considered. Therefore we break the complexity down to correspond to the three BFT operation phases: Notify, prepare and commit.

8.3.2.1 Notify Phase With regards to the notify phase, the best case scenario is that the message is received by one replica only. This would result in the receiving replica relaying the transaction to other replicas, which means that the best-case message complexity for the notify phase is $n-1$. If the client sends the same transaction to all replicas, and if the replicas had not received a notify message concerning the transaction, then the worst-case for the notify phase is that all replicas send the same notify message to all other replicas, i.e. the worst-case message complexity for this phase is $(n-1)^2$. However, if the replicas had already received a notify, they will simply discard the client's transaction, since replicas would have already started the BFT operations and had saved the transaction locally.

8.3.2.2 Prepare Phase With regards to the prepare phase, replicas send prepare messages if the replica votes that the client owns the coins included in the transaction. Normal operations, where a client owns the coins in a transaction and replica's will send their vote in a prepare message, requires a message complexity of $n \times (n-1)$ for this phase.

8.3.2.3 Commit Phase The commit phase also includes similar costs to the prepare phase. If replicas receive enough prepare votes, they will generate a commit message and send it to all replicas in the network. Therefore, the message complexity for this phase is also $n \times (n-1)$.

8.3.3 Verification Transactions Message Complexity

The normal-case scenario for the verification procedure is that the client awaits a response from a single replica, and that replica is non-faulty and relays the commit proof (including various other replicas' messages with their valid signatures). Therefore, the message complexity in this case is 1. However if the client asks for proof from all replicas, which is acceptable if the client wants to verify the status of the transaction on all replicas rather than just one replica, then the client would send a transaction status request to all replicas. This would lead to a worst-case message complexity of n messages.

8.3.4 Total Transaction Message Complexity

From our previous analysis of message complexity of different components in the system, we can add the results to provide an analysis of the total transaction message complexity (beginning with the client's transaction and ending with the transaction being executed, ownership of coins changed, and finally processing client verifications). We consider the normal-case operation in this subsection (and not the worst-cases that are due to malfunctions or malicious behaviour), to provide an idea of the communication costs incurred in our system.

The total message complexity, during normal operations, is given as $1+(n-1)+n\times(n-1)+n\times(n-1)+1 = 2(n)^2-n+1$ which we identify as **equation-1**. This means that for a normal transaction, and given the four replicas in our implementation, we would require $2(4)^2-4+1 = 29$ messages to be sent.

Given that a transaction message averages at 315 bytes (call this value a), replica messages average at 52 bytes[11] (call this value b), a notify message averages at 610 bytes (call this value c), and finally a client verification request along with a response from a replica with the transaction proof averages at 376 bytes (client request) + 1423 bytes (replica proof) = 1799 bytes (call this value d). Given **equation-1** above, this means we need to send the following amount of network traffic: $a+(b\times(n-1))+(c\times 2(n^2-n))+d$ which we identify as **equation-2**. Therefore, in our case where we deploy four replicas, we send $315+(52\times(4-1))+(610\times(2\times(4^2-4)))+1423 = 21406$ bytes of network traffic to complete a transaction (i.e. we need to send at least 20.9 KB per transaction). Note that this estimate does not include resending messages due to failure (which is handled by Python's `socket` TCP stream).

8.3.5 Propagation Delays

The equations and results discussed above show that the major cost for our system is message complexity. As discussed in *Appendix A5*, replicas were deployed on NeCTAR cloud in different locations; each location had a different network bandwidth as shown in Table 2. Given the information shown in Table 2, we estimate the network propagation delay based on the slowest speed allocated to replica4, with a maximum network bandwidth of 1.15 MB/s. Given **equation** -2 discussed in the previous subsection, the network propagation delay (for the worst-case message complexity, where we need to send at least 21406 bytes) is calculated as follows (**equation**-3): $t\_propogation$ = **equation**-2 / network bandwidth (in bytes) = $[a+(b\times(n-1))+(c\times 2(n^2-n)+d]/$ [network bandwidth (in bytes)] $= 21406 \div (1.15\times 1024\times 1024) = 0.01775$ seconds. This means that if we relocated replica4 in a data centre with a higher or similar bandwidth to other replicas, we are likely to get better results. However, we do not prioritise getting faster results over simulating a practical and real-world Internet environment. Moreover, although network bandwidth speeds of 14.5 MB/s are not uncommon, replica4's bandwidth speed of 1.15 MB/s may better reflect speeds of international links (given network latency, hop count, per-hop process delay, and other network transmission delays). Therefore, we chose not to change replica4's location and slower bandwidth, and base our estimate on its network performance.

Replica Name	**Bandwidth Speed**
replica1	14.5 MB/s
replica2	13.8 MB/s
replica3	14.5 MB/s
replica4	1.15 MB/s

Table 2 - Replicas' network bandwidth speeds

Given the calculations above, the running time scales with the number of replicas n. In **equation**-2, particularly the part concerning consensus messages ($c \times 2(n^2 - n)$), increasing the number of replicas will significantly increase network propagation delays. Therefore, to limit the propagation delay to a maximum of 0.05 seconds, we try values for n in **equation**-3: $[a + (b \times (n-1)) + (c \times 2(n^2 - n)) + d] \div (1.15 \times 1024 \times 1024) = 0.05$. Therefore, to achieve 0.05 propagation delay, n must be limited to 7 replicas (the result is almost 0.0513 seconds for propagation delay time). Increasing the total number of replicas n to 7 replicas results in the implementation being able to withstand f=2 faulty replicas (since f = (number of replicas $-1) \div 3$), while incurring more communication costs as outlined above (due to increasing n).

[11] Replica messages' average size are significantly less, since we send the hash digest of a transaction, instead of the entire transaction, during BFT operations.

8.4 Power consumption

We estimate the power consumption in Australia at AUD 0.2458 per kWh (AGL energy estimate). Since the implementation is designed to be deployed on replicas in different location, then the power consumption can be lowered even further if replicas are implemented in countries with low energy costs. However, we aim to give a general idea regarding the power consumption of our implementation. Given that an average server consumes about 362W per hour [66], to calculate the power consumption of four replicas for 30 days = $4 \times (362/1000) \times 24 \times 30 \times 0.2458$ = AUD 256.26 (almost USD 238.32). This means that our system uses AUD 8.54 (almost USD 8) per day for all four replicas to operate.

8.5 Comparison with Bitcoin

In the following subsection, we provide an analysis of Bitcoin's performance and results averaged for the month of June 2014. Bitcoin has reached 12878450 at the time of writing, and we aimed to generate the same amount of coins to produce an accurate comparison. It is important to mention that scaling the results to match Bitcoin's with regards to miners' performance is a complicated task and is a research in itself; it is difficult to determine how many miners there are in the network, let alone determine each miner's CPU performance and energy costs. Therefore, we provide a comparison that discusses Bitcoin's recorded performance with regards to transaction time and disk usage, as well as provide a conservative and optimistic estimate of Bitcoin's power consumption that can be used as a baseline for future research. Table 3 provides a summary of the comparison of the main features in Bitcoin and our implementation.

8.5.1 Bitcoin Transaction Time

Bitcoin average transaction time (over a period of 30 days) = 7.2677 minutes = 436 seconds/ transaction (transaction times obtained from blockchain.info). This means that Bitcoin can generate 0.0023 transactions/sec. Moreover, the Bitcoin network requires that clients wait for more blocks to be generated to ensure a 51% attack is not possible to invalidate the transaction, which means that clients need to wait for up to 60 minutes (6 blocks, at 10 minutes per block) to minimise the possibility that their transaction is reversed. Moreover, sending a Bitcoin transaction to the network does not guarantee that it will be included in the first block, a client may need to wait for the next block or more for the transaction to be confirmed in at least one block. However, we provide Bitcoin with a best-case scenario of requiring only one block to confirm a transaction, and accept 7.3 minutes per transaction to be Bitcoin's best performance.

8.5.2 Bitcoin Disk Usage

At the time of writing, Bitcoin's global ledger (blockchain) reached the size of 18,255 MB in size. The full size is required for verifying transaction since the genesis block. However, a process called *pruning* is described in the original Bitcoin paper [56], that reduced the size of the ledger so that only critical data required for verification is necessary. However, the *pruning* method is not implemented in the Bitcoin network yet, and the full ledger size is being used. Moreover, the *pruning* method assumed that some transactions will be marked as "fully spent", i.e. they are completed and cannot be changed. Due to the nature of Bitcoin's consensus, it is difficult to mark transactions as fully spent since they can be reversed by another longer blockchain and, thus, transaction finality is questionable (refer to *Literature Review* section). This could explain why the pruning method is not implemented yet, as it would probably require manual intervention, thereby compromising the network's supposedly decentralised consensus model.

8.5.3 Bitcoin Power Consumption

Estimating power consumption costs in the Bitcoin network is a complex task, since there exists various miner hardware devices with varying hash-rates and energy costs. Moreover, as indicated earlier when comparing our system's power consumption, Bitcoin miners can deploy their mining equipment in countries with lower energy costs, so the estimate using Australian energy costs may be relatively high. However, we will provide a very conservative and optimistic estimate for Bitcoin's power consumption, by calculating the total consumption using a popular mining device called Monarch, produced by Butterfly labs[12], with the following specification: 600 GH/s, 350 W per hour (indicated as "conservative estimate" on the product information page), and priced at USD 2,196. Given that the Bitcoin network requires an average 81367108.6 GH/s (June 2014), and if we were to replicate the required hash rate using Monarch devices, we would require 81367108.6 / 600 = 135612 Monarch devices. A single Monarch device uses (350/1000) × 0.2458 = AUD 0.08603 worth of electricity per day. Estimating Bitcoin's power consumption comes to about 135612×0.08603 = AUD 11666.7 per day. This result is conservative since the Monarch device is a PCI card that relies on other equipment to supply it with power and consume considerable energy. Moreover, we have not factored in the total cost of buying 135612 Monarch devices (which comes to a total of USD 297,803,952). However, we accept this conservative estimate as Bitcoin's total power consumption, since even the most optimistic and conservative estimates show that Bitcoin is a total waste of energy.

Feature	**Bitcoin**	**Our System**
Transactions/sec (average)	0.0023	2.97
Transaction/block (average)	394	NA
Transactions/ Day (average) *	56736	256608
Energy Costs / Day (in AUD)	11,666.7	8.54
Number of Servers	NA	4
Cost/transaction (in AUD)	0.2	0.000024
Disk Usage (12.878 million coins)	18,255 MB	663 MB

Table 3 - Comparison of main features between Bitcoin and our system

* Bitcoin's value is calculated by multiplying Transaction/block (average, listed above) × number of blocks per day (144 block a day)

8.6 Threat Scenarios

In the following subsections, we provide a brief analysis of how certain attacks are thwarted in our implementation. We discuss Denial of Service attacks and our prevention mechanisms, as well as how the system handles double-spending attempts.

8.6.1 Denial of Service

Our implementation is capable of logging malicious attempts and IP addresses (ephemerally in memory), and blocking clients for a period of time then reaccepting their transactions after that time passes. We provide a discussion of how a replica handles invalid signatures. First the implementation is designed to process spam and rapid transactions, using a counter for invalid attempts. Function **process_spam()** is called to increase a client's invalid attempts' counter.

[12]Information obtained from http://www.butterflylabs.com/monarch/

If a client's signature fails validation, then a replica logs an error as shown in Listing 8, and calls **process_spam()**:

```
[05/07/14 11:00:19] replica1 [CRYPT|ERR] invalid client signature from
  172.16.129.127:56813
[05/07/14 11:00:19] replica1 [EXP] invalid client message from 172.16.129.127:56813
```

Listing 8: Replica log showing invalid client signature

If **process_spam()** determines that the client has exceeded the threshold for transaction attempts (currently set at 100 transactions per 5 seconds), the client is blocked for 5 seconds, and all connections from that IP address are dropped immediately, as shown in Listing 9:

```
[05/07/14 11:00:19] replica1 [CONN] connection from 172.16.129.127:56861
[05/07/14 11:00:19] replica1 [IP] dropping connection from 172.16.129.127
```

Listing 9: Replica log showing client connections are dropped

Moreover, It is possible to enhance this procedure to output the IP block list to a simple text file, or extract it from the replica's log to be used with `iptables` in Linux to drop connections at the kernel (Network Layer) for faster connection filtering speeds.

8.6.2 Double-spending

Recall that double-spending is the malicious attempt to pay different entities with the same coin. In our system, this scenario would translate to the client attempting to send two or more transactions for the same coin. Once a transaction for that coin is executed, then the client does not own the coin anymore, and subsequent attempts are discarded and flagged as spam (a call to **process_spam()** is made). First we show that the transaction on the coin is successfully executed in Listing 10:

```
[06/11/14 07:05:47] replica3 [BYZ] transaction fbb732_fa includes 1 coins. Processing
  each coin, if any coin is not owned by the client I will discard the transaction.
[06/11/14 07:05:48] replica3 [BYZ|FIN] successfully executed transaction fbb732_fa
```

Listing 10: Replica log showing successful execution of the initial transaction

Then, after attempting to restore the same `clientcoin.data` (client's coin database) to pay with the same coin, the client's transaction is rejected[13]. The result is shown in Listing 11.

```
[06/11/14 07:08:30] replica3 [TRN|ERR] check ownership result: client does not own
  coin 1
[06/11/14 07:08:30] replica3 [BYZ] my coinTable shows that the user does not own
  coin(s) included in transaction 2b3bbc_b0 , ceasing to process this transaction; will
  only prepare and commit based on replica votes
```

Listing 11: Replica log showing the double-spend attempt is blocked

[13]The hash digest of the transaction is different from the original attempt. This is due to the timestamp included in the transaction, which changes the result of the hash digest.

9 Challenges and Limitations

In this chapter, we discuss the challenges that may face the implementation in a production environment. We also discuss the limitation of what properties the current implementation is incapable of providing. We focus mainly on security aspects in our analysis.

9.1 Server-side Security

The issue of securing replicas' private keys must be given careful consideration. Private keys are currently stored unprotected in the software repository, since we assume other measures are taken to isolate and protect the hosts running the software implementation. However, an adversary who steals the private keys can compromise the integrity of the consensus: An adversary can then manipulate the system to successfully execute double-spending attacks (issue verification of transactions for the same coin to different payees). As an example of how trivial it is to compromise private keys if care is not taken to isolate the hosts: The implementation was hosted on NeCTAR's cloud, and the system administrators who grant virtual hosts to researchers are likely capable of taking snapshots of the servers. This means that they can then extract the private keys from disk and compromise the system.

9.2 Threshold Cryptography

Although we had considered threshold cryptography in our initial planning, we decided not to implement it due to the overhead it would add to the implementation. Threshold cryptography can be critical to prevent malicious behaviour by forcing replicas to cooperate to unlock and use a shared private key. It might be possible to modify this implementation to remove individual replica signatures, and replace those operations with threshold cryptography. The modification might produce an increase in latency and consensus times, since replicas would need to wait for other cooperating and non-faulty replicas to use a shared private key.

9.3 Denial of Service

Although we have implemented a Denial of Service (DoS) prevention component in our implementation, a DoS attack is still a major threat to our implementation. Due to the nature of the implementation's distributed topology, it is easier to target the replicas than it would be in a decentralised topology. We aimed to minimise this threat through our DoS prevention component, yet we expect a large DoS attack may harm the implementation if other prevention techniques are not implemented.

9.4 Replica Membership

The current implementation does not allow for dynamic replica membership. In fact, the process is executed manually, and we performed the short membership procedure at the initial stage of deployment. However, the protocol suffers from the inability to dynamically allow new nodes to join the closed membership of replicas. We have not devised a plan to provide such dynamic membership. More research is required for the practicality of a dynamic membership protocol for our implementation. Laurie's blueprint [48], suggested the dynamic membership of replicas, and [27] might provide the steps required to provide a robust implementation of such a protocol.

10 Conclusion

Through this research, we aimed to create a practical solution for a distributed currency. We explored Bitcoin's features and weaknesses; we adopted its strength with regards to cryptographic operations while aiming to eliminate its inefficient consensus protocol. We explored Byzantine Fault Tolerance (BFT) literature to create an alternative distributed consensus to Bitcoin's computationally expensive Proof-of-Work (PoW) consensus. We realised that most BFT implementations aimed for high throughput while neglecting to focus on maintaining system progress.

Therefore, we created a BFT implementation using a broadcast model; to send replies to all other replicas, and let replicas decide what action to take next based on other replicas' votes, without requiring a leader (which could have slowed down the implementation substantially). We built a full implementation from scratch using Python, and used our BFT consensus component to reach agreement regarding transactions. We presented a protocol for the monetary component of our distributed payment system, which handles transactions, verifications and minting. We also provided evaluation analysis of the current system implementation, as well as challenges and limitations that can be addressed to provide proactive security and add resilience to the implementation.

We aimed to provide a more efficient and elegant solution to consensus and the unregulated synchronisation of the global ledger. We eliminated Bitcoin's brute-force method for consensus that requires solving computationally expensive puzzles, and relies on the majority of computational power to be owned by honest nodes in the network (a requirement even a single entity can violate). Rather, we require a majority number of honest nodes to operate in an implementation we prepared for a hostile Byzantine environment, where a replica cannot trust other replicas and is prepared to recover from Byzantine behaviour. Transactions processed in our system are almost instantaneous, final, and irreversible. Therefore, we believe that we have achieved the goals of our project by providing a usable implementation for providing an alternative to centralised financial systems—without requiring third parties or central authorities—which can outperform Bitcoin's recorded average performance, using a fraction of the energy wasted in Bitcoin, and at a fraction of its energy costs.

References

[1] M. Babaioff, S. Dobzinski, S. Oren, and A. Zohar. On bitcoin and red balloons. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, pages 56–73. ACM, 2012.

[2] A. Back et al. Hashcash-a denial of service counter-measure, 2002.

[3] R. Baldoni and J. Hélary. A component-based methodology to adapt the fault tolerance of distributed protocols.

[4] R. Baldoni, J.-M. Hélary, M. Raynal, and L. Tangui. Consensus in byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, 2003.

[5] T. Bamert, C. Decker, L. Elsen, R. Wattenhofer, and S. Welten. Have a snack, pay with bitcoins. 2013.

[6] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming.* Springer, 2011.

[7] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology—Crypto 2001*, pages 524–541. Springer, 2001.

[8] M. Castro and B. Liskov. Authenticated byzantine fault tolerance without public-key cryptography. Technical report, Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[9] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[10] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[11] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[12] C. Clark. Bitcoin: A technical introduction. 2013.

[13] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Position paper: Bft: the time is now.

[14] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[15] M. Correia, N. F. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

[16] M. Correia, G. S. Veronese, N. F. Neves, and P. Verissimo. Byzantine consensus in asynchronous message-passing systems: a survey. *International Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

[17] L. A. de la Porte. The bitcoin transaction system. 2012.

[18] C. Delporte-Gallet, H. Fauconnier, and A. Tielmann. Fault-tolerant consensus in unknown and anonymous networks. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 368–375. IEEE, 2009.

[19] J. Dinger and H. Hartenstein. Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration. In *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 8–pp. IEEE, 2006.

[20] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[21] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[22] J. R. Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

[23] D. Drainville. An analysis of the bitcoin electronic cash system, 2012.

[24] P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.

[25] N. Ferguson and B. Schneier. *Practical cryptography*, volume 23. Wiley New York, 2003.

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

[27] O. Garcia Morchon, H. Baldus, T. Heer, and K. Wehrle. Cooperative security in distributed sensor networks. In *Collaborative Computing: Networking, Applications and Worksharing, 2007. CollaborateCom 2007. International Conference on*, pages 96–105. IEEE, 2007.

[28] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[29] L. Gong, P. Lincoln, and J. Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. *DEPENDABLE COMPUTING AND FAULT TOLERANT SYSTEMS*, 10:139–158, 1998.

[30] J. N. Gray. *Notes on data base operating systems*. Springer, 1978.

[31] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding [distributed fault tolerant systems]. In *Distributed Computing Systems, 1997., Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of*, pages 183–188. IEEE, 1997.

[32] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama. Performance comparison between the paxos and chandra-toueg consensus algorithms. In *Proc. of International Arab Conference on Information Technology, Doha, Qatar*, pages 526–533, 2002.

[33] M. Herrmann. *Implementation, evaluation and detection of a doublespend-attack on Bitcoin*. PhD thesis, Master Thesis ETH Zürich, 2012, 2012.

[34] G. Karame, E. Androulaki, and S. Capkun. Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin. *IACR Cryptology ePrint Archive*, 2012:248, 2012.

[35] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *OPODIS*, volume 97, pages 61–75. Citeseer, 1997.

[36] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper. Jpaxos: State machine replication based on the paxos protocol. *Faculté Informatique et Communications, EPFL, Tech. Rep*, 167765, 2011.

[37] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[38] J. A. Kroll, I. C. Davey, and E. W. Felten. The economics of bitcoin mining or, bitcoin in the presence of adversaries.

[39] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[40] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[41] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[42] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23. Springer, 2003.

[43] L. Lamport. Leaderless byzantine consensus, Sept. 14 2010. US Patent 7,797,457.

[44] L. Lamport. Byzantizing paxos by refinement. In *Distributed Computing*, pages 211–224. Springer, 2011.

[45] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[46] B. Lampson. The abcd's of paxos. In *PODC*, volume 1, page 13, 2001.

[47] B. Laurie. Decentralised currencies are probably impossible but let's at least make them efficient. *Practice*, 100(11):1–10, 2011.

[48] B. Laurie. An efficient distributed currency. *Practice*, 100, 2011.

[49] B. Laurie and R. Clayton. Proof-of-work" proves not to work; version 0.2. In *Workshop on Economics and Information, Security*, 2004.

[50] N. Lesperance. Distributed consensus in byzantine systems. 2012.

[51] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[52] P. J. Marandi, M. Primi, and F. Pedone. Multi-ring paxos. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[53] H. Meling, K. Marzullo, and A. Mei. When you don't trust clients: Byzantine proposer fast paxos. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 193–202. IEEE, 2012.

[54] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 2010.

[55] A. Miller and J. J. LaViola Jr. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin.

[56] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1:2012, 2008.

[57] S. Nakamoto. Re: Bitcoin p2p e-cash paper, 2008.

[58] C. Paar and J. Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer, 2010.

[59] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[60] M. Rosenfeld. Analysis of hashrate-based double-spending. 2012.

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

[62] B. Schneier. Applied cryptography. protocols, algorithms, and source code in c/bruce schneier, 1996.

[63] Y. J. Song, R. Van Renesse, F. B. Schneider, and D. Dolev. The building blocks of consensus. In *Distributed Computing and Networking*, pages 54–72. Springer, 2008.

[64] D. J. Sorin. Fault tolerant computer architecture. *Synthesis Lectures on Computer Architecture*, 4(1):1–104, 2009.

[65] P. Urbán and A. Schiper. Evaluating the performance of distributed agreement algorithms. Technical report.

[66] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah. Worth their watts?-an empirical study of datacenter servers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10. IEEE, 2010.

[67] Z. YE, S. LI, X. ZHOU, and J. ZHOU. An overview of consensus in asynchronous distributed systems. *Journal of Computational Information Systems*, 8(10):4047–4060, 2012.

Appendix

A1: Python implementation of a simplified Proof-of-Work (PoW)

```
#import python modules
import random, hashlib, hmac
import sys, pickle, os, base64, re, time, binascii
from Crypto.Hash import SHA512, SHA256, SHA
from subprocess import call
from binascii import hexlify
from multiprocessing import Process

#average block size, 17KB
#for a simplified version, we don't consider
#or hash the total blockchain size
cont = os.urandom(17000)

#number of 0s as hash prefix
d = 4

def pow():
    counter =0
    #nonce 32 bits, 4 bytes
    nonce =os.urandom(4)
    #hash the value
    x = hashlib.sha256(cont+nonce).hexdigest()
    #check the outcome if it contains a certain prefix
    #of 0's required by the set difficulty in variable d.
    #loop until prefix is found in result.
    while (int(x[0:d],16) !=0):
        nonce =os.urandom(4)
        x = hashlib.sha256(cont+nonce).hexdigest()
        #uncomment the following line to print results
        #print x
        counter +=1
    return counter, x, nonce

counter , x, nonce = pow()
print counter, x, nonce
print nonce.encode('hex')
x = hashlib.sha256(cont+str(nonce)).hexdigest()
print x
```

A2: PBFT Operations

The following sections include a summary (with some modifications) of the major components of PBFT [9] that we aim to implement in our system. We aimed to simplify and summarise PBFT, and we added a few comments to simply understanding.

Consensus Operations

Upon receiving a request from a client, the role of replicas (including the *primary*) is to verify that the message is not malformed (otherwise it is discarded). Messages with invalid signatures, whereby the public key attached to the message does not verify the client's signature, are also immediately discarded. If the message is well-formed and its signature is valid, a replica logs $D(m)$ for tracking and performance monitoring purposes. The primary p is chosen as $p = v \bmod \mathsf{R}$, where v is the current view number R is the set of replicas. If the receiving replica is not the primary, a *backup* (non-primary replica) signs the received message after validation, and forwards the message to the primary. Signatures are used as proof that backups stand by their verification, and to provide evidence in case of Byzantine behaviour.

The primary follows the same procedure detailed above (verify form of message and signature) after receiving a message (either relayed by a backup or directly sent from a client). Reprocessing of the message is necessary to monitor backups' behaviour and to flag Byzantine replicas that send invalid messages. The primary creates $D(m)$ before verification, and if it is already processing m, it can quickly discard any subsequent messages regarding m without having to validate those messages again. If it has not logged $D(m)$ before, then it verifies m as outlined above. At this stage the primary is ready to start the three-phase consensus protocol of PBFT as outlined in [9].

Pre-Prepare Phase

The primary creates sequential numbering for messages: The primary assigns the number n to the client's message, and the PRE-PREPARE message (along with the client's request m) is broadcasted to all backups. This message is inserted into the primary's log in the following format: $\langle\langle$ PRE-PREPARE, v, n, $D(m)\rangle_{\sigma_p}$, $m\rangle$, where v indicates the view (stage) in which the message is being sent, m is the client's transaction, and $D(m)$ is m's digest. Note that if the client's request has been relayed, backups have logged $D(m)$ and expect a broadcast regarding the transaction in a timely fashion. This assists in the process of monitoring the primary's performance; delayed transaction broadcasts are a sign of Byzantine behaviour or degraded performance and warrant view changes. A backup accepts a PRE-PREPARE message if the following elements are correct:

i. The signatures are valid, and $D(m)$ is indeed the hash digest of m. The primary signs the hash of m, and inserts the signed hash into the PRE-PREPARE message. Replicas can verify the hash and the signature since m will be attached to the PRE-PREPARE message.

ii. The backup is in the correct view as relayed by the primary (value of v is indeed the backup's current view).

iii. It has not already accepted a PRE-PREPARE message with the same sequence number n and view v.

iv. The sequence number n is between a lower bound h and higher bound H. This is required to prevent sequence number depletion, and will be explained in a later section (see *garbage collection*).

Prepare Phase

Upon receiving a PRE-PREPARE message from the primary, a backup validates the request. If it accepts the request after checking the properties listed above, it enter the PREPARE phase, otherwise it discards the message. In the PREPARE phase, a backup multicasts a PREPARE message to all replicas (including the primary) with the following format: $\langle$PREPARE, v, n, $D(m)$, i $\rangle_{\sigma_i}$. Note that the backup does not send m, since replicas should have already received a copy of m in the PRE-PREPARE phase. A replica adds the PREPARE message to its log if the following properties are satisfied:

i. The signature is valid.

ii. The replica is in the correct view as indicated by the PREPARE message.

iii. The sequence number n is between a lower bound h and higher bound H (see *garbage collection*).

Next, all replicas wait for votes regarding the transaction. The transaction's status is changed to *prepared*(m, v, n, i) if a replica i appended the following to its log:

i. The transaction m (relayed by the primary in PRE-PREPARE phase).

ii. A PRE-PREPARE entry for m, in view v, with sequence n.

iii. The replica received $2f$ PREPARE messages (votes) from other replicas concerning m.

The replicas must validate that the PRE-PREPARE entries match the PREPARE entries by checking that the matching entries have the same view, sequence and hash digest. This last step creates a total ordering of transactions within the system's current view.

Commit Phase

When a replica gathers enough votes concerning a transaction m, and the transaction's status has been converted to prepared as outlined above, then the COMMIT phase is initiated by broadcasting a message in the following format: $\langle$COMMIT, v, n, $D(m)$, $i\rangle_{\sigma_i}$. Replicas must verify COMMIT messages before inserting the entry into their logs. The properties to be checked are similar to those for validating PREPARE messages outlined above (valid signatures, correct view, and correctly bounded sequence number).

In order to ensure the execution of transaction in order, the following properties are defined and must be verified. A transaction is *committed*(m, v, n) iff *prepared*(m, v, n, i) is true for all i in set of f+1 non-Byzantine replicas. Moreover, a transaction is *committed-local*(m, v, n, i) iff *prepared*(m, v, n, i) is true and replica i accepted $2f$+1 valid COMMIT messages from other replicas (including itself) that match the PRE-PREPARE messages (same view, sequence, and hash digest). Therefore, granted the checks are valid, this ensures that if a *committed-local* is valid for a non-Byzantine node i, then *committed* is also valid. This mechanism guarantees that non-Byzantine replicas agree on the sequence of transactions committed locally even if they are not committed in the same view. It also guarantees that a *committed-local* status of a transaction will ultimately be *committed* at least f+1 non-Byzantine replicas.

A replica executes m after *committed-local* is true and replica i has already executed transactions with lower sequence numbers. This process is needed to satisfy the requirement of executing transactions in order. After the execution of a transaction, a replica sends its result to the primary in the form: $\langle$REPLY, v, t, *address*(c), i, $r\rangle_{\sigma_i}$, where v is the view number, t is the transaction timestamp, *address*(c) is the address of the client who sent the transaction, i is

the replica's address (or possibly its preconfigured numbering in the set R), and r is the reply value which is the whole coin to identify its full status. The primary waits for $f+1$ valid replies with the same t and r to accept the result. These $f+1$ replies (or votes) constitute the result of the consensus protocol and is the result of the client's transaction. The primary then relays the information to the client, who can check the $f+1$ replies to verify they are valid results sent by different replicas.

Garbage Collection

We now describe how replicas maintain an agreement regarding the state of the system. This process, called *garbage collection*, is required to maintain proof about the correctness of the system's state, which progresses the system and can be used to discard earlier messages. Moreover, this process can provide replicas with a verifiable state in case a replica requests information after a crash.

To produce a checkpoint, a replica broadcasts a $\langle \mathsf{CHECKPOINT}, n, d, i \rangle_{\sigma_i}$ to all other replicas, where n is the last sequence number of the transaction executed in the state for which a checkpoint is being produced, and d is the digest of the system state (the log of executed transactions). We can possibly send d as a vector of hash digests of each transaction executed since the last *stable* checkpoint (a checkpoint with proof messages is a called *stable*). Replicas must collect CHECKPOINT messages as proof of a checkpoint's stability, until $2f+1$ such messages (including its own) are collected. These signed messages are verifiable and dictate the state of the system at a certain checkpoint. From that point onwards, replicas need not process messages with sequence numbers less than or equal to n. Therefore, all PRE-PREPARE, PREPARE and COMMIT messages with such sequence numbers, as well as earlier CHECKPOINT messages, can be discarded from a replica's logs.

Generating checkpoints includes the process of agreeing on a low and high bound for sequence numbers (as mentioned in the three phases of the consensus operations). These bounds control what messages will be accepted to prevent a Byzantine primary from abnormally depleting sequence numbers in order to confuse backups. The low bound h is equal to the sequence number of the last transaction executed (used in generating the checkpoint as explained above). The high bound $H = h + k$, where k is chosen to be high enough in order to prevent stalling the system while waiting for stable checkpoints to be generated. Therefore, the value of k depends on how frequently we generate checkpoints. It is impractical to generate stable checkpoints too frequently, e.g. after every transaction, since that will stall the system until a checkpoint is generated, and this will cause substantial overhead. If checkpoints are created every 1000 transactions, then we can choose k to be 2000 to allow for a window big enough to generate a checkpoint before the next checkpoint must be generated.

View Change

The view change process is a critical component that ensures liveness and progress. This process is used when the primary's performance is questionable. Recall that backups log transactions they receive and await execution based on the primary's directions. Backups start a timer after logging a transaction (if a timer is not already running for a transaction), and the timer is reset if the transaction is executed (and possibly started again for another transaction). If a timer of any replica expires in a view v, then that replica starts the view change procedure to move the system to view $v+1$ and replace the current primary.

During a view change, replicas that want to change the view stop accepting messages regarding transactions (they accept messages relating to checkpoints and view changes). Replica i

broadcasts a $\langle$VIEW-CHANGE, $v{+}1$, n, C, P, $i\rangle_{\sigma_i}$ message, where n is the last sequence number in the last stable checkpoint s in i's records, C is the $2f{+}1$ valid checkpoint messages that prove s is valid (see *garbage collection*), and P is the set of transactions prepared at i with sequence numbers higher than n, but have not been executed yet. Each transaction must include valid PRE-PREPARE message and $2f$ matching and valid PREPARE messages signed by various backups (with the same view, sequence and hash digest), to prove the validity of the transaction as the system moves towards a new view. Therefore, valid and signed VIEW-CHANGE votes from various replicas ($2f$ of such messages) must be broadcasted before the system moves towards a new view. A single faulty replica, or a quorum of less than $2f$ faulty replicas, cannot start view changes.

When a new primary p in view $v{+}1$ receives $2f$ valid view change messages for the view $v{+}1$ (recall that primary p is chosen as $p = v \bmod \mathsf{R}$, where R is the set of replicas), it broadcasts $\langle$NEW-VIEW, $v{+}1$, V, $O\rangle_{\sigma_p}$, V is the set containing valid and signed VIEW-CHANGE messages (as explained above), and O is the set of PRE-PREPARE messages generated as follows:

i. The primary determines the lowest sequence (*min-s*) of the most current stable checkpoint in V, and the highest sequence (*max-s*) in a transaction's PREPARE message in V.

ii. The primary must now create new PRE-PREPARE messages for view $v{+}1$ for each n between *min-s* and *max-s*. This step has two cases:

   (a) There is a set in P contained in V with the previous requirement regarding n: The primary creates a new $\langle$PRE-PREPARE, $v{+}1$, n, $D(m)\rangle_{\sigma_p}$, where $D(m)$ is the transaction digest included in the PRE-PREPARE message for n with the highest view number in V (replicas use $D(m)$ as a reference to m which they had already stored).

   (b) There is no set P that satisfied the above requirement. The primary creates a new $\langle$PRE-PREPARE, $v{+}1$, n, $D(null)\rangle_{\sigma_p}$, where $D(null)$ is the digest of a null request which is processed normally as a transaction but its execution is a *no-op*.

The primary then logs set O. If *min-s* is greater than the sequence number in the current primary's checkpoint, then the primary generates a new checkpoint with *min-s* as the last sequence number, and discards information (as explained in *garbage collection*). The primary now enters view $v{+}1$ and accepts transactions for that view. Backups validate new view messages (if signed and messages are valid for $v{+}1$) by validating the set O; the process of validating the set O is similar to the process followed by the primary to create the set. Backups then move towards stabilising the system by broadcasting PRE-PREPARE messages for each transaction in O, logging the transactions, and entering view $v{+}1$. The system now proceeds with the normal protocol (as explained in *Consensus Operations*).

Although view changes must be implemented in order to react to a Byzantine primary, the substantial overhead and degraded performance of the system during such critical phases require careful consideration. View changes effectively stall system progress: A new primary must assume the leading role to process buffered transactions, replicas must authenticate the messages sent by the new primary, and replicas which issue view change messages stop accepting transactions. Therefore, depending on the system implementation and evaluation, we may need to implement the optimisations discussed in [14, 37] to prevent a complete halt of system progress.

A3: Practical Implementation Details

A3-1 Test Environment

To test the software implementation on a private network before deployment on a cloud infrastructure, we created virtual nodes: 4 server nodes (to meet the minimum for the consensus threshold, i.e. when f is at least 1, then the number of servers $n = 3f+1 = 4$) and 2 client nodes. The server nodes (*replicas*) as well as the client nodes ran on FreeBSD10 which was chosen for its stability and minimal resources requirement (4 replicas and 2 clients ran on two cores and only 256MB of RAM each). The lab environment was hosted on Mac OSX, which was used to develop the software using Xcode. Furthermore, the software implementation was tested on Mac OSX, Ubuntu, Debian, and FreeBSD. Therefore, it is likely that any Unix or Linux OS equipped with Python 2.7.* can run the software. The implementations has not been tested on Windows yet; we suspect that minor changes need to be implemented with regards to getting the local IP address of the host.

A3-2 Cryptographic Operations

We sign and encrypt every network transmission in our system. Elliptic Curve Cryptography (ECC) was implemented using a publicly available python library, PyECC[1], which is based on the Digital Signature Standard (DSS), also known as FIPS 186-3 standard. PyECC uses FIPS approved curves, and we chose the curve P-256 and the corresponding parameters for our implementation to provide a compromise between security and performance. We chose PyECC since it requires no installation by the user, which facilities deployment and enhances the usability of the client program. We include the details of the elliptic curve, including its parameters, in *Appendix A4*. Furthermore, we modified PyECC to use SHA256 as the default hash digest algorithm (the default was SHA1). Cryptographic operations for our implementation is included within `cryptops.py`, whereas PyECC's cryptographic operations are included in `eccrypt.py` and `ecdsa.py` (within `ecc` folder). Note that in the example included in the following subsections, we use a single key pair for demonstration. In practice, we use two separate key pairs for sender and receiver.

ECC Keys

In order to obtain the public key, a random integer is first generated, which constitutes the private key. This private key d is used to perform ECC Point Multiplication with the curve's Generator G, to obtain a point on the curve; this point is the user's public key Q: `dG = Q`. The interface to create the key pairs is shown below (**gen_ecc()** in our implementation, calls **keypair()** function from `ecdsa.py` found in `ecc` folder).

[1] Obtained from https://github.com/amintos/PyECC

def gen_ecc(*bits*):

input: *bits*: number of bits

output: *keypair* : list data type containing two lists: the first list contains the public key, the second list contains the private key. The first element of each list represents the number of the bits used to generate the key.

function: The purpose of this function is to choose a random number d, to be used to generate a public key $Q{=}dG$ using EC point multiplication. The resultant is a keypair containing two list types: the public key is in the first list, and the private key in the second list. The first element of each list contains the number of bits.

Thus, the format of the keypair is `((bits, (x,y)), (bits, d))` and an example is shown below:

```
>>> from ecdsa import keypair
>>> keypair(256)
```

Output

```
((256, (34191731306973987224869480145737876327479497655138088305517884236652075273485L,
46876348624286984804857271818719216245302475051743365080928734668776336768 44L)),
(256, 49505765503873642628818666616948943051769847903740300002745844270249569481724L))
```

The public key `pubk` and the private key `prik` are stored along with the number of bits, and are obtained from the double list `keypair` above:

```
pubk = keypair[0]
prik = keypair[1]
```

Encryption and Decryption

PyECC does not encrypt the full contents of the data using the ECC public key. Rather, it uses the Rabbit stream cipher to encrypt the data, using a shared secret key. The process involves generating k randomly (random integer between 1 and n, where n is the order of the curve). A temporary public key kG is generated, where G which is the curve's generator point. Then, a shared secret sG is generated by multiplying s with the receiver's public key Q, $sG = k(Q){=}k(dG)$. The shared secret key's X-coordinate is then used as input to the **encypter()** function (Rabbit algorithm) to generate the cipher-text. The sender transmits kG to the other party. The recipient can then compute the same shared secret $sG = k(dG){=}(kG)d$ (recall that d is the user's private key, known only to her). The recipient can then use sG's X-coordinate for decryption. This method of exchanging the shared secret is known as Elliptic Curve Diffie–Hellman Key Exchange (ECDH-KE) algorithm for exchanging keys. In our implementation, we transmit the cipher-text along with the temporary key kG, since extracting k from kG is difficult according to the Elliptic Curve Discrete Logarithm Problem (ECDLP), and this is the essence of ECC.

The functions encrypt and decrypt can be found in PyECC's `eccrypt.py` file (within `ecc` folder in our software implementation). We provide below the interface for interacting with PyECC to provide encryption and decryption (we omit the signature and verification methods).

def encrypt(M, Q, *encrypter* = *Rabbit*):

input: M: the contents to be encrypted,
Q: public key
encrypter: default symmetric algorithm function for encryption (the only and default option is Rabbit)

output: C : cipher-text version of M
kG : temporary key

function: The purpose of this function is to choose a random number k, to be used to generate a temporary key kG and a shared secret key sG= k(Q)= k(dG). sG is used as the encryption key for the *Rabbit* cipher algorithm, and the output consists of the cipher-text C and the temporary key kG.

An example of using the encryption function:

```
>>> from eccrypt import *
>>> encrypt("test message",
(256,
(34191731306973987224869480145737876327479497655138088305517884236652075273485L,
468763486242869848048572718187192162453024750517433650809287346687763367684 4L)))
```

Output

```
('\x10Wjx&nd\xf33\xc17M',
(10248615414473359996287586420033691489472268849840170012571089738244441537 9519L,
43492214852114705839520707302644983210662235442903641213976749216439526855132L))
```

where `'\x10Wjx&nd\xf33\xc17M'` is the ciphertext, which we will use in the example for the decryption function, and the second item is the list containing the X- and Y-coordinates of the public key Q.

def decrypt(C, kG, d, *encrypter* = *Rabbit*):

input: C : cipher-text version of M
kG : temporary key
d : user's private key
encrypter: default symmetric algorithm function for encryption (the only and default option is Rabbit)

output: M : plain-text version of C

function: The purpose of this function is to calculate the shared secret key sG = k(dG)=(kG)d. sG is used as the decryption key for the *Rabbit* cipher algorithm, and the output consists of the plain-text M.

An example of using the encryption function (using the cipher-text above):

```
>>> decrypt('\x10Wjx&nd\xf33\xc17M',
 (10248615414473359996287586420033691489472268849840170012571089738244441537 9519L,
```

```
434922148521147058395207073026449832106622354429036412139767492164395268551 32L),
(256,
4950576550387364262881866661694894305176984790374030000274584427024956948 1724L))
```

Output

```
'test message'
```

Hash Function

Before we can discuss signatures and the verification process, we need to discuss the hash digest function used in the signature function. We created a simple function called **hashthis()**, which takes any content as input, and outputs the SHA256 hash digest of the content. The implementation uses Python's built-in module `hashlib` to create hash digests. PyECC includes a similar function that combined the signature and verification procedures, yet we implemented a separate function since we frequently create hash digests of data for tracking or logging (without necessarily signing the hash digest). We include the interface of the hash function, as well as two examples: The first shows how to use hash digests within our implementation (using the **hashthis()** function), and the second shows how to create hash digests in any Python implementation.

Example of **hashthis()**:

```
>>> from cryptops import hashthis
>>> hashthis("test message")
```

Output

```
'3f0a377ba0a4a460ecb616f6507ce0d8cfa3e704025d4fda3ed0c5ca05468728'
```

Example of using hash digest in Python (this is called within **hashthis()**):

```
>>> import hashlib
>>> hashlib.sha256("test message").hexdigest()
```

Output

```
'3f0a377ba0a4a460ecb616f6507ce0d8cfa3e704025d4fda3ed0c5ca05468728'
```

The two examples above can be verified in a terminal:

```
% /usr/bin/perl -e "print qq(test message)" | shasum -a 256
```

Output

```
3f0a377ba0a4a460ecb616f6507ce0d8cfa3e704025d4fda3ed0c5ca05468728  -
```

Furthermore, we created a function called **short_id()** to generate a short ID of SHA256 hash digests to simply logging, tracking and verification (with regards to creating a human-readable and short versions of the hash digests). In other words, instead of visually comparing the entire 64-character hash digest above and using it for logging, we use **short_id()** to concatenate the first 6 character of the hash digest, with a single underscore and the last two characters of the hash digest. Note that we only generate short IDs for logging, we compare the full hash digest of the content rather than the short ID. If we used short IDs (for comparing content), it would defeat the purpose of hash digest algorithms, as it can significantly increase hash collisions. The **short_id()** output appears frequently in the implementation log, since we use **short_id()** for transaction tracking and logging. Therefore, it was necessary to document this function to assist in understanding system operations.

An example is shown below:

```
>>> from main import short_id
>>> short_id(
```

```
"3f0a377ba0a4a460ecb616f6507ce0d8cfa3e704025d4fda3ed0c5ca05468728")
```
Output
```
'3f0a37_28'
```

Moreover, within the address generation function, RIPE160 hash function is used as well: It generates 160 bit hash digest instead of 256 bits which is the result of SHA256.

Signatures and Verification
To create digital signature, we sign the hash digest of the content to be signed, rather than the content itself. Therefore, **hash_sign()** function, found in PyECC's `ecdsa.py`, creates a hash digest of the content m passed to it as the first parameter (we modified the code to use SHA256 rather than SHA1), then signs the resultant hash digest using the second parameter which is the private key d. We created a **create_signature()** function that first validates d, then calls **hash_sign()** to generate a signature *signM* of the content. We show below the interface and an example of using signatures.

def create_signature(m, d):

input: m : data to be signed
d : private key

output: *signM* : signature of the hash digest of m

function: The purpose of this function is to validate the private key d, then create a SHA256 hash digest h of the content m, and finally sign h and return the signature *signM*.

An example of using the **create_signature()** function:
```
>>> from cryptops import create_signature
>>> d = (256 ,
49505765503873642628818666616948943051769847903740300002745844270249569481724L)
>>> create_signature("test message",d)
```
Output
```
('sha256',
389635631710674779500112693641049869096483510460197816619626216088694404707 02L,
358014933293596602146779108789907436057009279741835572573856165591991762957 52L)
```

The other entity can then use the signer's public key Q to verify the signature. The recipient validates the public key passed as the third parameter to the function, generates their own version of the hash digest h of m passed as the first parameter, and verify the signature *signM* passed as the second parameter. The output is a boolean *True* or *False*.

def verify_signature(m, *signM*, Q):

input: m : data to be verified,
signM : signature of hash digest h of m,
Q : signer's public key

output: *Boolean Value*

function: The purpose of this function is to validate the public key Q, then create a SHA256 hash digest h of the content m, and finally verify h using Q. The output is a Boolean value *True* or *False* representing the result of the verification.

```
>>> from cryptops import verify_signature
>>> verify_signature("test message", ('sha256',
 38963563171067477950011269364104986909648351046019781661962621608869440470702L,
358014933293596602146779108789907436057009279741835572573856165591991762957 52L),
(256,
(34191731306973987224869480145737876327479497655138088305517884236652075273485L,
46876348624286984804857271818719216245302475051743365080928734668776336768 44L)))
```

Output

```
True
```

A3-3 Generating Addresses

We implemented Bitcoin address generation method from scratch to create addresses for clients and replicas, and we used the publicly available `base58.py` code[2], to convert the result of the hash digest to its final representation in `Base58`. In the following subsections, we discuss the main interface for **init_client()** which creates ECC key pairs (as discussed in the aforementioned sections) as well as the address (from the ECC public key). We discuss how we generated addresses to be unique for our test network, and easily identifiable.

Address Creation

We generated addresses specifically for our test network, by using a constant 1-byte prefix (`0xff`) to represent the first byte. The result is that all addresses start with `12m` or `12n` and are easily verifiable and identified for their validity for our test network (simple substring match). Given a tuple (i.e. array or list) `kpub` as an ECC public key, we follow the following tasks to create a `Base58` address:

1. Extract (x, y):

```
# the tuple is in the following format: (256, (x,y))
# we need the second element, hence kpub[1],
x = kpub[1][0]
y = kpub[1][1]
```

2. Add the network prefix to create *raw_address_str*:

```
raw_address_str = address_const_prefix + str(x) + str(y)
```

3. Create a SHA256 hash digest *address_sha256* in hexadecimal format of *raw_address_str*:

```
address_sha256 = hashlib.sha256(binascii.unhexlify(raw_address_str)).hexdigest()
```

4. Create a RIPE160 hash digest of *address_sha256*, we need to use Python's `binascii unhexlify` module to get the byte representation of data:

```
        h = hashlib.new('ripemd160')
        h.update(binascii.unhexlify(address_sha256))
        address_ripe = h.hexdigest()
```

5. Prepend the network prefix:

[2] Obtained from https://github.com/sirk390/coinpy

```
net_address_ripe = network_type_prefix+address_ripe
```

6. Create SHA256 hash digest of the result above, then another digest using the same hash function to produce `sha256sha256_net_address_ripe`:

```
shasum_a = hashlib.sha256(binascii.unhexlify(net_address_ripe)).hexdigest()
shasum_b = hashlib.sha256(binascii.unhexlify(shasum_a)).hexdigest()
sha256sha256_net_address_ripe = shasum_b
```

7. Use the first 4 bytes as the address checksum (8 characters), and prepend the checksum to the result obtained in step 5:

```
address_checksum = sha256sha256_net_address_ripe[0:8]
net_address_ripe_checksum = net_address_ripe + address_checksum
```

8. Finally, encode the result obtained in step 7 using `Base58`, and return the result as the address:

```
address = base58test.base58encode(int(str(net_address_ripe_checksum),16),1)
return address
```

Encoding the address uses `Base58` to avoid using visually ambiguous characters (such as the letter 'O' and the number '0'). The following characters are used to encode the result of `Base58`:

```
b58chars = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'
```

As mentioned earlier, specifying a constant network prefix results in addresses starting with `12m` or `12n`. Address have the following format (`base58`):

```
12mr5KkjEeyjE4BTJMZ7XVuGXUmkprgobVuq
```

An example of using **gen_address()** to is shown below:

```
>>> from cryptops import gen_address
>>> gen_address((256,
(139178561741835167032622694325695130875817414869349464556752801359422486869 74L,
77534501911246108522457546065461191713081279652025574823263185781989530580742L)))
```

Output

```
'12mwM2NzN8PKqFfs6W12AQp6fTnqWMCB2FBy'
```

Testing Address Creation

To test that our address generation implementation is correct, we used the website http://brainwallet.org/ to get public keys and store it as *raw_address_str* in our **gen_address()** function above. This matches the created **test_address(***raw_address_str***)** function we created specifically to test address generation. We include below screenshots obtain from the aforementioned website, as well as matching results from the **test_address()** function (which is similar to **gen_address()** discussed above, but does not prepend the network prefix constant, since it is already included in the input).

Matching results shown using `test_address`:

```
>>> from cryptops import test_address
>>>
test_address("0478d430274f8c5ec1321338151e9f27f4c676a008bdf8638d07c0b6be9ab35c71a1518
        063243acd4dfe96b66e3f2ec8013c8e072cd09b3834a19f81f659cc3455")
```

Output

```
        '1JwSSubhmg6iPtRjtyqhUYYH7bZg6htxqm'
```

Note that the address prefix is '`1`' rather than '`12m`' or '`12n`' as implemented in our test network. The prefix '`1`' is the prefix of Bitcoin addresses and is due to using `\x04` for the first byte instead of `\xff`.

Main Interface

The main interface function **init_client()** is used to generate replica information prior to deployment, as well as new information for clients. It calls **gen_ecc()** from PyECC to create key pairs, as well as **gen_address()** discussed earlier.

def init_client():

input: None

output: *keypair* : list data type containing two lists, the first list contains the public key, the second list contains the private key. In both lists, the first element is the number of bits

address : address generated from public key *kpub*

function: The purpose of this function is to generate *keypair* using **gen_ecc()** function, the public key *kpub* is then obtained from *keypair*`[0]` (i.e. the first element), and passed to **gen_address()** which creates the user's address in `Base58`.

An example of generating client or replica information using **init_client()** results in the following data (*pubk*, *prik*, *address*):

```
>>> from cryptops import init_client
>>> init_client()
```

Output

```
(((256,
 (139178561741835167032622694325695130875817414869349464556752801359422486869 74L,
775345019112461085224575460654611917130812796520255748232631857819895305807 42L)),
(256,
6180297803648484946031512292668361664781842444323105215861391298570605919534L)),
'12mwM2NzN8PKqFfs6W12AQp6fTnqWMCB2FBy')
```

A3-4 Database Operations

We used hash tables to store data in memory for fast retrieval and processing. Python's built-in `dictionary` data type, can be used to simulate hash table operations. Each dictionary entry has a key and an optional value. In our implementation, we create a hash digest of the content to produce the key of the dictionary value. We demonstrate how dictionaries can be used in

Python: We first create an empty dictionary called *Mail* (initialisation is unnecessary in Python, but shown to simplify understanding), assign text to m, create a hash digest h of m, and using h as the key to store the contents of m. Then we print the whole contents of *Mail*, but the important component of this demonstration is the final step of passing h as a parameter to *Mail* and obtaining only the value stored in *Mail*[h]:

```
>>> Mail={}
>>> m="this is a test message"
>>> from cryptops import hashthis
>>> h=hashthis(m)
>>> Mail[h]=m
>>> print Mail
{'4e4aa09b6d80efbd684e80f54a70c1d8605625c3380f4cb012b32644a002b5be':
'this is a test message'}
>>> print Mail[h]
this is a test message
```

Dictionaries can include any data type, and we even use dictionaries to store dictionaries within them or tuple data types (arrays). We refer to dictionaries as hash tables in the following sections.

Hash Table Search

In order to search a hash table for a key, we use the built-in **has_key()** function on a dictionary to obtain a Boolean value. The following code shows how to search for m above (given *Mail* was already processed as demonstrated earlier):

```
>>> m="this is a test message"
>>> h=hashthis(m)
>>> print Mail.has_key(h)
True
>>> m="wrong message"
>>> h=hashthis(m)
>>> print Mail.has_key(h)
False
```

Hash Table Insert

To insert content we create a hash digest of the contents and pass the hash to the dictionary as a key, and the content as the value. It is important to note that if the key already exists in the hash table, the contents will be replace with the new data. We are aware of this issue, and use it to update contents in various hash tables in our implementation. The code below shows how to insert content in a hash table, as well as ensuring the key does not exist to avoid replacing existing content:

```
>>> m="new message"
>>> h=hashthis(m)
>>> if not Mail.has_key(h):
...     Mail[h]=m
...     print "stored new m"
... else:
...     print "I already have this message"
...
stored new m
```

Hash Table Delete

The operation to delete content from a hash table requires passing the key that must be deleted (the contents will be deleted as well). There are methods to pass values instead of keys to delete content, but we omit those methods for brevity. The code below demonstrates how to use hash tables to delete a key and its content:

```
>>> print Mail
{'821a8de6ff88501123b99cacf7eec178638c77362012f73945f21b624cd105f8': 'new message',
'4e4aa09b6d80efbd684e80f54a70c1d8605625c3380f4cb012b32644a002b5be': 'this is a test
message'}
>>> m="this is a test message"
>>> h=hashthis(m)
>>> del Mail[h]
>>> print Mail
{'821a8de6ff88501123b99cacf7eec178638c77362012f73945f21b624cd105f8': 'new message'}
```

Python Pickle Module

We use Python's built-in `pickle` and `cPickle` modules to write data to disk, and load data as `pickle` objects into memory. `pickle` is also used to serialise the data (using `pickle.dumps,` not to be confused with `pickle.dump` used for writing data to files). We use the module extensively to store `*.data` files in our implementation (more importantly `coin.data` and `clientcoin.data`). Using `pickle`, we load information as objects rather than processing files line by line. This significantly speeds up load times and writing data to disk, as well as eliminates the need to manually serialise the data (reading each element and writing it on a line, reading the next element and doing the same, and so on). Therefore, we use `pickle` to write lists and dictionaries as objects. We include below the interface for writing data using `pickle.dump`; we omit the load function since it is similar, yet it uses `pickle.load` instead of `pickle.dump` to load objects and restore their original format.

def dump_table_overwrite(*table*, *file*):

input: *table* : hash table (dictionary) to be written to disk
file : name of file to write *table* contents into

output: modified *file* : contents of table are written in `pickle` object format in *file*.

function: The purpose of this function is to write content of variable (in `pickle` object format) to *file* (which can be a list, dictionary, or any other object that can be serialised, i.e. data types that have indices). This function uses `pickle.dump` to serialise objects.

The following code demonstrated a simplified example of how `pickle` can be used to write and serialise objects:

```
>>> import pickle
>>> m=[(0,1),[2,4]]
>>> print m
[(0, 1), [2, 4]]
>>> testfile="tf.txt"
>>> dumpfile = open(testfile, "wb")
>>> pickle.dump(m,dumpfile)
```

(*view content using a terminal*)

```
% cat tf.txt
(lp0
(I0
I1
tp1
a(lp2
I2
aI4
```

A3-5 Network Operation

We use Python's built-in modules `SocketServer` and `socket` to implement a networking layer for our implementation. The layer is responsible for delivering data over the network, and it is not involved in modifying or processing data. Three major components of the implementation, found in `netops.py`, are implemented in this layer: **init_socket()**, **recv_data()** and **send_data()**. We provide the interface for the three functions below.

def init_socket():

input: None

output: *s*: Stream (TCP) socket

function: The purpose of this function is to create a socket *s* listener using Python's `socket` module; *s* is a stream (TCP) socket, and *s* is configured to reuse the address in order to create multiple simultaneous connections.

def recv_data(*s*, *n*):

input: *s*: TCP socket, which is created using **init_socket()** as described above.
n : an integer representing how many bytes should be accepted into *s*.

output: *data*: *n* bytes of data received into the socket

function: The purpose of this function is to receive *n* bytes of data into *s* and return the received *data*.

def send_data(*data*, *sender_ip, sender_port, recipient_ip, recipient_port, conn*):

input: *data*: information to be sent
sender_ip: IP of sender (host's IP)
sender_port: Port of sender (host's IP)
recipient_ip: IP of recipient
recipient_port: Port of recipient
conn: socket object

output: *Boolean* variable indicating whether *data* was sent over the network successfully or sending failed (if a connection cannot be established).

function: The purpose of this function is to send *data* given the parameters passed to it (IPs and ports). Although we use the same port for *sender_port* in our implementation, we maintain this parameter in case of future modifications where different ports can be used.

Threading

A fourth critical component of our networking operations is the ability to process multiple connections, and not wait until a connection ends to process another. We use Python's built-in `threading` module to create a C class for `ClientThread` objects; this class uses the given parameters (*ip_address*, *port*, *conn*) to initialise the object. We provide below the interface to the `ClientThread` C class.

class ClientThread(*thread.Thread*):

input: *Thread* : call to `threading` model to prepare new thread of object
(*input to class call*)
ip_address: Address of remote host
port: Port of remote host
conn: socket object (for *ip_address* and *port*)

output: None, a class is created and a **serverops()** call is initiated (discussed below).

Function: The purpose of this function is to create a thread object to handle multiple connections and to start **serverops()** (the main operations in a server node, found in `server.py`, for processing networking information and calling various components for Byzantine Fault Tolerance operations).

We show below an example of using `threading` and `ClientThread` class. Note that `newthread` is appended to the `threads` component of `threading,` and *addr* represents a two-element list in the following format (*ip*, *port*):

```
newthread = ClientThread(addr[0], addr[1], conn)
newthread.start()
threads.append(newthread)
```

TCP stream sockets

We chose to implement TCP stream sockets to gain the connection-oriented benefits of TCP over the speed and unreliability of UDP. We require reliable sending and packet reassembly in case of fragmentation or errors in transmissions; this is necessary to process encrypted data that must be reassembled correctly at the destination, and packets are retransmitted in case of errors or failures in connections. Moreover, servers listen for connections on TCP/62176, and as mentioned in **recv_data()**, sockets listen for a specific number of bytes before ending the connection (if the number of bytes is exceeded, a connection is dropped and the message is discarded). We configured the socket to listen to 20000 simultaneous connections. Python

documentation recommends listening to a reasonable power-of-2 bytes (i.e. 2^n bytes), and we chose a relatively high value (4096 bytes) in order to receive large amounts of data due to processing of large dictionaries (hash tables) that must be sent to other replicas to validate and commit transactions, and are sent to clients to provide proof for transaction verifications.

A3-6 Miscellaneous Components

DoS Prevention

Blocking spam using a hash table for IPs and monitoring repeated errors or rapid requests.

Encrypted Email Implementation

We implemented an encrypted email function, allowing users to send and receive emails, querying replicas for users' public keys and checking that the relayed public key produces the same **gen_address()** result (verifying that the public key is indeed the intended recipient's public key). The email message is signed then encrypted using the recipient's public key, then encrypted again with the replica's public key (multiple versions are created since we send to all replicas). An example is shown below.

Sender Side

```
Enter the user's address ( example: 12mqHaYUnCqaStkFJ13hmJDPQ6JySYQ4TbjP )
>12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL
[05/25/14 15:19:38] client [MAIL|CRYPT] contacting replica1 to get public key for
12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL
[05/25/14 15:19:38] client [SND] sending to 172.16.129.201:62172
[05/25/14 15:19:38] client [SND|FIN] sent to 172.16.129.201:62172
[05/25/14 15:19:38] client [MAIL|CRYPT] valid replica signature and public key
properties received. Proceeding to process key for
12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL

----- [PUBLIC KEY RECEIVED] -----
[Address] 12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL
[Hash] e529f7c52f76c6453a611c6b02508094bdd4778476148bd16a7fbcdcabace809
[Fingerprint] e5:29:f7:c5:2f:dc:ab:ac:e8:09
---------------------------------------

----------- [WARNING] ------------
Verify the fingerprint shown above with the recipient

If you want to accept this fingerprint and store this key, type the number shown    below:
(for now, typing enter will also proceed in beta testing phase)
---------------------------------------

Enter this to continue ---> 9490

>9490

[05/25/14 15:19:50] client [MAIL|CRYPT] stored public key for
12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL locally as e5:29:f7:c5:2f:dc:ab:ac:e8:09
[05/25/14 15:19:50] client [INFO] storing pubkeys.data

Enter your message
[Press Enter without typing anything to end entry]
>test message
```

```
>

[FOR] 12mvaNcYYBG1HyRApPFSykN333WZvP26i7yL
-----BEGIN UNENCRYPTED DATA-----

test message


-------END UNENCRYPTED DATA-------

Press Enter to encrypt data...

-----BEGIN SIGNED ENCRYPTED DATA-----

gAJdcQAoVZPAIk35RIoEpSDPpp5V17idF9O
+iXzKXPtRWP2r10cP7d3llkH1OjPrFDWUgbTVy7YRltPhkEnyKgYW5E52uuIWAABa4XpYI8uWUey9oz7KWERo
roVL4T/Hg5XCtGjm/A3IW3Owj6xGkfMtmG+dOlkgyqsK9b0V3B4ZVN8Pb0s/
3Me9I2pMYW3zs4efmipL7r7qktVxAYogxe8hZydqvjLwfQE2QpzFJfLyRbg7rsm6GrjL50Q8xkWKIXMGgqPhB
BIYtOrOivIxZMp+0VwHwhdRxpJsne53vZmgAIZxAmUu

-----END SIGNED ENCRYPTED DATA-----

Press Enter to send encrypted message...
```

Recipient Side

```
-----BEGIN ENCRYPTED MESSAGE-----

gAJdcQAoVZPAIk35RIoEpSDPpp5V17idF9O
+iXzKXPtRWP2r10cP7d3llkH1OjPrFDWUgbTVy7YRltPhkEnyKgYW5E52uuIWAABa4XpYI8uWUey9oz7KWERo
roVL4T/Hg5XCtGjm/A3IW3Owj6xGkfMtmG+dOlkgyqsK9b0V3B4ZVN8Pb0s/
3Me9I2pMYW3zs4efmipL7r7qktVxAYogxe8hZydqvjLwfQE2QpzFJfLyRbg7rsm6GrjL50Q8xkWKIXMGgqPhB
BIYtOrOivIxZMp+0VwHwhdRxpJsne53vZmgAIZxAmUu

-----END ENCRYPTED MESSAGE-----

[05/25/14 15:22:46] client2 [MAIL|CRYPT] contacting replica1 to get public key for
12mr5KkjEeyjE4BTJMZ7XVuGXUmkprgobVuq
[05/25/14 15:22:46] client2 [SND] sending to 172.16.129.201:62172
[05/25/14 15:22:46] client2 [SND|FIN] sent to 172.16.129.201:62172
[05/25/14 15:22:46] client2 [MAIL|CRYPT] valid replica signature and public key properties
received. Proceeding to process key for 12mr5KkjEeyjE4BTJMZ7XVuGXUmkprgobVuq

-----BEGIN DECRYPTED MESSAGE-----

test message


-----END DECRYPTED MESSAGE-----
```

A4: Elliptic Curve Cryptography Parameters

We include below the details of the Elliptic Curve used in our implementation to provide cryptographic operations. This information is obtained from the PyECC source code documentation. We show up to 256-bit curve parameters, since we use ECC P-256 for our implementation.

```
#
#   Predefined Elliptic Curves
#   for use in signing and key exchange
#
'''
Predefined elliptic curves for use in signing and key exchange.
This Module implements FIPS approved standard curves P-192, P-224, P-256,
P-384 and P-521 along with two weak non-standard curves of field size 128
and 160 bits.

The weak curves cannot be used for signing but provide a faster way to
obfuscate non-critical transmissions.
'''
# FIPS approved elliptic curves over prime fields
# (see FIPS 186-3, Appendix D.1.2)
DOMAINS = {

  # Bits : (p, order of E(GF(P)), parameter b, base point x, base point y)

  192 : (0xfffffffffffffffffffffffffffffffeffffffffffffffL,
      0xffffffffffffffffffffffff99def836146bc9b1b4d22831L,
      0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1L,
      0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012L,
      0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811L),

  224 : (0xffffffffffffffffffffffffffffffff000000000000000000000001L,
      0xffffffffffffffffffffffffffff16a2e0b8f03e13dd29455c5c2a3dL,
      0xb4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4L,
      0xb70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21L,
      0xbd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34L),

  256 : (0xffffffff00000001000000000000000000000000ffffffffffffffffffffffffL,
      0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551L,
      0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604bL,
      0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296L,
      0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5L),

CURVE_P = 3     # global parameter of all curves (for efficiency reasons)
```

A5: NeCTAR Cloud Implementation

We deployed our implementation on NeCTAR research cloud. The implementation consisted of fours servers designated as replicas as shown in Figure 10. We chose to deploy replicas in different locations, as indicated by the *IP address* and *Zone* in Figure 10, to simulate an Internet environment where network latency affects communication.

Instance Name	Image Name	IP Address	Zone	Size
replica3	NeCTAR Ubuntu 14.04 (Trusty) amd64	130.56.250.7	NCI	m1.medium | 8GB RAM | 2 VCPU | 10.0GB Disk
replica4	NeCTAR Ubuntu 14.04 (Trusty) amd64	130.220.208.109	sa	m1.medium | 8GB RAM | 2 VCPU | 10.0GB Disk
replica2	NeCTAR Ubuntu 14.04 (Trusty) amd64	118.138.240.246	monash-01	m1.medium | 8GB RAM | 2 VCPU | 10.0GB Disk
replica1	NeCTAR Ubuntu 14.04 (Trusty) amd64	130.56.248.93	NCI	m1.medium | 8GB RAM | 2 VCPU | 10.0GB Disk

Figure 10: Replica cloud deployment information

As shown in the figure above, we used NeCTAR's Ubuntu 14.04 linux distribution. The custom NeCTAR Ubuntu image was pre-configured for SSH connectivity, along with many other features which made deployment of server a seamless procedure.

A6: User Documentation

The information included below is intended to help users familiarise themselves with the software implementation. Brief operations and explanations are provided for those interested in running replica (server) operations, as well as regular users (client operations).

```
For server/replica operations:

1. You must create server keys and information using "gen-server-info.py"
At a terminal, navigate to the source directory (same directory you found
this README.txt in):

$ python gen-server-info.py

This will generate the server's required information (we will call each server a "replica")
The previous step (using gen-server-info.py) must be performed on all replicas
to generate their keys and information.

One replica, or all, can then create the replicainfo.data (which stores replica
information necessary for the system to operate). This means that the software
looks for replicainfo.data to know how to contact replicas (network operations), and its public
     key (for encrypting) messages.


2. Run "update-replica-table.py" on one of the replicas:

$ python update-replica-table.py

Now enter the information of each replica as prompted. This will generate the required file "
     replicainfo.data"

This file must now be saved at every replica (i.e. place the same replicainfo.data
into src/data/ for each replica). Now replicas know how to communicate with other replicas.

3. Run the server node

$ python server.py

The implementation is configured to mint if it is running for the first time,
so you'll see nodes trying to communicate to other replicas. If they are all running
and can communicate with each other, they are likely to successfully mint. However,
this is not always the case. If minting is successful then one of the replicas earned
a reward. To claim this reward switch to client operations.

4. Copy src to another location (creating two repositories, one for replica, one for
client is advisable). Although you can run the client and server from the same src
repository, I recommend you separate the two in case you overwrite something (the
file coin.data in "data" folder holds the system's global ledger, overwriting it
means your replica is faulty, and will be ignored by non-faulty replicas)

-------Go to 5 if you want to claim your reward-------

5. If you won a reward (check your local replica_mint.log, to know who is the recipient of
the mint reward), then you can only claim it by running client operations.
You will need the same keys used by your replica to claim the reward. Copy the folder
to another location and follow the steps outlined in "For client operations" below.
--------------------------------------------------------------------------------
For client operations:

1. Use a terminal window to navigate to "src" folder (where you found this README.txt file)
```

2. In the terminal window, type the following:

$ python client.py

You will be able to select various options, and likely need to generate your keys
and address (unless you are moving from server/replica operations to client operations).

3. Generate your "Create address and keypairs" if the prompt suggests that you should
(i.e. if you don’t see your address and other user information, create new keypairs).

4. If you have coins, you can spend them. If you don’t, you can prompt replicas
to ask them if you were paid any coins. (if you are a replica that won a mint vote,
you claim your reward by selecting the "Get Payments" option.

--- That’s it for client operations, it’s straight forward from here if you are
comfortable using a terminal window ---

Notes:
a. If you want to use the Email feature, the other user you want to communicate with
must have communicated with the replicas already (otherwise replicas do not know the
other user?s information). This can easily be done if the other person merely chooses
"Get payments" even if they know they will not have any payments. The reason this works:
When you contact replicas, you’re encrypting the message using that replica’s public key,
so other cannot read your message. At the same time, and this is the important part,
you’re sending ∗your∗ public key in that message. Your public key is what is needed
for secure communication (using the Email feature).

b. ∗Careful∗ not to overwrite your existing keys. You will not be able to spend your
coins if you don’t have the keys (or overwrite them), and payments sent to that
address/key cannot be spent anymore, by you or anyone else.