FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring semester, 2018 Open/~~Confidential~~
Author: Racin W. Nygaard	Racin W. Nygaard (signature of author)
Programme coordinator: Hein Meling Supervisor(s): Hein Meling, Leander Jehl	
Title of master's thesis: Distributed Storage with Strong Data Integrity based on Blockchain Mechanisms	
Credits: 30	
Keywords: Blockchain, Storage System, Distributed Systems	Number of pages: 103 + supplemental material/other: Code included in PDF Stavanger, June 15, 2018

Distributed Storage with Strong Data Integrity based on Blockchain Mechanisms

Racin W. Nygaard

June 15, 2018

Abstract

A blockchain is a datastructure that is an append-only chain of blocks. Each block contains a set of transaction and has a cryptographic link back to its predecessor. The cryptographic link serves to protect the integrity of the blockchain. A key property of blockchain systems is that it allows mutually distrusting entities to reach consensus over a unique order in which transactions are appended. The most common usage of blockchains is in cryptocurrencies such as Bitcoin.

In this thesis we use blockchain technology to design a scalable architecture for a storage system that can provide strong data integrity and ensure the permanent availability of the data. We study recent literature in blockchain and cryptography to identify the desired characteristics of such a system. In comparison to similar systems, we are able to gain increased performance by designing ours around a permissioned blockchain, allowing only a predefined set of nodes to write to the ledger. A prototype of the system is built on top of existing open-source software. An experimental evaluation using different quorum sizes of the prototype is also presented.

Acknowledgements

I would like to thank Cristina Nistor, my family and my good colleagues at Qualisoft for all their support during my Master's degree.

I would also like to thank Professor Hein Meling for supervising the thesis and Leander Jehl for providing valuable feedback.

Contents

List of Figures

List of Tables

Abbreviations

ABCI Application blockchain interface

API Application programming interface

BFT Byzantine fault tolerance

CA Certificate authority

CAS Content-addressable storage

CID Content identifier

DAG Directed acyclic graph

DER Distinguished encoding rules

DHT Distributed hash table

DLS Dwork Lynch StockMeyer

DoS Denial-of-service

gRPC gRPC Remote Procedure Calls

HVT Homomorphic verifiable tag

IPFS Interplanetary file system

MAC Message authentication code

P2P Peer-to-peer

PBFT Practical byzantine fault tolerance

PKCS Public-key cryptography standards

PoET Proof-of-elapsed-time

PoS Proof-of-stake

PoSt Proof-of-storage

PoW Proof-of-work

RSA Rivest Shamir Adelman

SFS Self-certifying file system

SMR State machine replication

Nomenclature

fork Two or more blocks at the same height.

mempool A cache of transactions in memory before inclusion in blocks.

miners Specialized nodes which partake in mining.

mining Solving computationally hard puzzles in order to validate and record new transactions on the ledger. Is the security mechanism in proof-of-work consensus.

Chapter 1

Introduction

The introduction of Bitcoin and its continuous coverage in mainstream media has brought a lot of attention to the underlying technology, blockchains. Blockchains are viewed as being a driving force for innovation in the modern digital society. There is on-going work in several areas for the use of blockchains, such as the financial market, electronic voting, tracking data provenance in clinical trials, data storage and many others.

A blockchain is an append-only distributed ledger of transactions, whose structure is kept resilient through the use of cryptographic primitives. Thus, the study of a blockchain system is an interdisciplinary endeavor, involving key concepts from mainly cryptography and distributed systems. The strength of the technology is more apparent when used in a distributed fashion, where it allows consensus to be reached between parties that do not trust one another. Much of the research on blockchains is focused on the consensus mechanisms. These range from the rediscovery of algorithms such as in the case of proof-of-work, to more traditional methods such as Byzantine fault tolerance, and even new novel methods such as proof-of-space/time. The choice of consensus mechanism is perhaps the most important element when considering a blockchain based system. Due to each mechanism offering different properties, it is necessary to carefully consider the specific intended application before making a choice.

An interesting use case for blockchains is to support a storage network of mutually distrusting entities. Key properties of a reliable storage system are scalability, reliability, throughput and availability. By using a blockchain to keep track of the stored data, additional properties such as verifiability of the data stored and linked timestamping [1, 3] can be gained. These properties

combined with cryptographic primitives will give the end-user the power to verify data authenticity and provide confidence in the permanent availability of the stored data.

The overall aim for this thesis is to design a highly scalable architecture for a storage system built on blockchain technology that requires strong data integrity. We foresee our system most suitable for applications which require public verifiability. Example applications include health journals, business contracts, scientific data and patents. To this end, we study recent literature in order to identify and classify the characteristics needed for the system. In our literature study, we identify three similar systems that rely on a publicly available ledger, which requires significant overhead to control the network. We are able to significantly reduce the overhead by designing our system to only allow a consortium of writers, which in addition allows us to benefit from traditional Byzantine fault tolerance algorithms.

The architecture is organized into three components; *clients*, *ledger nodes* and *storage nodes*. We separate the agreement protocol from the storage servers [43, 44] and study different quorum sizes for these two components. A prototype is developed and evaluated against the identified criteria. In our experiments we investigate the limitations of the prototype.

In addition, we investigate the following research questions:

RQ1 How can the ledger support high update and query rates, yet offer sufficient integrity guarantees?

RQ2 How large peer-to-peer network sizes is reasonable to support?

RQ3 What are the challenges of a permissioned blockchain?

1.1 Contributions and Outline

In summary, in this thesis we make the following contributions

- Design of a highly scalable architecture for a storage system built on a permissioned blockchain.
- Identification and classification of characteristics of the storage system.
- Development of a prototype for all three components of the architecture.

- Evaluation and experimental results of the prototype developed.

The remainder of the thesis is organized as follows:

Chapter 2 introduces relevant background material for reading the thesis. The focus will be on the most significant elements a blockchain system consist of.

Chapter 3 presents a survey of existing blockchain systems as well as works which relates to this.

Chapter 4 discuss of the design space and provides an overview of the system architecture.

Chapter 5 describes how a prototype of the system was implemented.

Chapter 6 evaluates the implementation and present a set of experimental results with different quorum sizes.

Chapter 7 presents suggestions for further work.

Chapter 8 concludes the thesis.

Chapter 2

Background

In this chapter we will describe some of the fundamental properties, concepts and terminologies needed to read this thesis and understand blockchain systems.

2.1 Consensus Protocols

The *consensus* problem lies at the core of distributed computing [10]. Processes use consensus to choose a single proposed value. Consensus can be expressed using the following *safety* and *liveness* properties [22, 23];

CS1 Only a value that has been proposed may be chosen.

CS2 Only a single value is chosen.

CS3 A process never learns that a value has been chosen unless it actually has been.

CL1 Some proposed value is eventually chosen.

CL2 Once a value is chosen, correct learners eventually learn it.

In this context, safety means to never violate the consistency of the shared state. If it is violated at some time t, it can never be satisfied again after that time. The liveness properties makes sure that given enough time, something advantageous will eventually happen. More formally, for any time t, there is a chance that the property will be satisfied at some time $t' \geq t$ [10].

The underlying consensus protocol of a distributed system is a mechanism to make the participating nodes agree on how the shared state will be updated. Each of the nodes may have their own copy of the shared state, and thereby increasing the availability of the system as a whole. Given the same initial state and by having all the nodes execute updates in the same order, the state will remain consistent across all nodes. This is known as state machine replication in the literature.

A consensus protocol guarantees the safety and liveness properties based on its system model, which specifies the environment it is designed for. The system model is typically described using assumptions about the elements in the system, such as: network links, the presence of synchronized clocks and the expected behavior of nodes. If the system model neglects to consider some real world aspect, one should not have confidence in the deployment of the protocol.

With regards to blockchain systems, the security and robustness [12] is largely based on the choice of consensus protocol. The type of consensus that is most relevant to discuss for blockchain systems is *atomic broadcast* [4], also known as *total order broadcast*. The next two sections will outline how assumptions about node behavior impact the consensus protocol.

2.1.1 Crash Tolerant

If nodes are prone to non-Byzantine faults, but otherwise follow the protocol, consensus can be reached if less than half of the nodes are faulty. The atomic broadcast abstraction has two events; *Broadcast* and *Deliver*. Broadcast is the request used when a node wants to send a message to the whole network. Deliver is the indication that outputs the message to the application layer. The abstraction guarantees global ordering on messages delivered for all the correct nodes, as long as only benign faults are present. The properties of atomic broadcast can be found in [10];

1. **Validity:** If a correct process p broadcasts a message m, then p eventually delivers m.

2. **No duplication:** No message is delivered more than once.

3. **No creation:** If a process delivers a message m with sender s, then m was previously broadcast by process s.

4. **Agreement:** If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

5. **Total order:** Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2, then q delivers m_1 before m_2.

One of the most well-known crash-tolerant consensus protocols is Paxos [22, 56], which requires $2f + 1$ nodes to tolerate f benign faults. In Paxos the network is compromised by three classes of agents; *proposers*, *acceptors* and *learners*. In a typical replicated state machine configuration, all nodes will take on the role as every agent during the protocol [16]. The clients do not partake in the consensus protocols, but will interact with proposers when issuing commands.

Paxos progresses in a sequence of rounds, where a single proposer is chosen to act as the *leader* for each round. Each round runs in two phases. The first phase is used to decide the leader for the round, and starts with the proposers selecting a round number r and sending it in a $\langle$PREPARE$\rangle$ message to the acceptors. If the round number r is higher than any previously seen, the acceptor will reply with a $\langle$PROMISE$\rangle$ message containing the highest-numbered proposal it has accepted.

The second phase starts once a proposer receives a response for round number r from a majority of the acceptors. The proposer will then broadcast a $\langle$ACCEPT$\rangle$ message containing the round number r as well a value v, where v is the value from the highest-numbered proposal amongst the responses. If no prior proposal was received, the proposer is free to use any value for v. The acceptor will accept the proposal for round number r and value v unless it has already replied with a $\langle$PROMISE$\rangle$ for a round greater than r. Upon accepting, the acceptor will broadcast a $\langle$LEARN$\rangle$ message containing the round number r and value v to all learners. Once a learner receives a message containing the same round number and value from a majority of the acceptors, the value is considered chosen.

2.1.2 Byzantine Fault Tolerant

Byzantine faults include both naturally occurring benign faults such as crashes and network partitioning, as well as maliciously behaving nodes. A node may be subverted by an invisible adversary which has the power to make it exhibit arbitrary behavior. For instance it may *equivocate* by sending conflicting

messages to different nodes, go out of protocol, control the scheduling of messages, issue *denial-of-service* attacks and otherwise degrade performance [57].

The problem was first illustrated in [11], in which a group of generals tries to agree on a battle plan by disseminating messages to each other. The difficulty lies in the fact that some of the generals may be traitors, and try to confuse the others. It is shown in [11] that it is possible to achieve consensus as long as the maximum number of byzantine faulty nodes in a network is less than a third of the total number of nodes.

It has been shown that deterministic protocols can not reach consensus in a fully asynchronous system [58]. This results may be circumvented in practical BFT protocols by adding randomness(e.g. cryptography), or making timing assumptions. The most well-known practical BFT protocol is PBFT [34], which requires $3f + 1$ nodes to tolerate f Byzantine faults. PBFT leverages message authenticators and assumes the partially synchronous model in which there is a unknown upper bound on message delivery time. Similarly to Paxos, PBFT progresses in a sequence of rounds, with a unique leader in every round [4].

Several techniques to reduce the redundancy requirements have been proposed [59, 44, 43, 60]. Particularly interesting for our protocol is the separation of agreement and execution phase [44, 43], where the replica requirement changes to $3f+1$ for the agreement and to $2f+1$ for executing state machine commands.

An example of a BFT protocol that works in an asynchronous network by adding randomness is HoneyBadgerBFT [60]. It adds randomness by a series of cryptographic techniques such as threshold signatures, erasure coding and secret sharing [60]. Experiments in [60] show a significant thoughput improvement over PBFT when the number of nodes is 16 or higher.

2.2 Blockchain

There is no standard technical definition of the term *blockchain* [5], however we will use the definition most commonly found in the literature, where it is defined as a data structure [1, 3]. The data structure is an ever-growing list of records, called *blocks*. The blocks are ordered in an append-only chain, where they have a cryptographic link to their predecessor, which servers to protect the blockchains integrity. It is not possible to remove or make changes to a block after it is added to the chain.

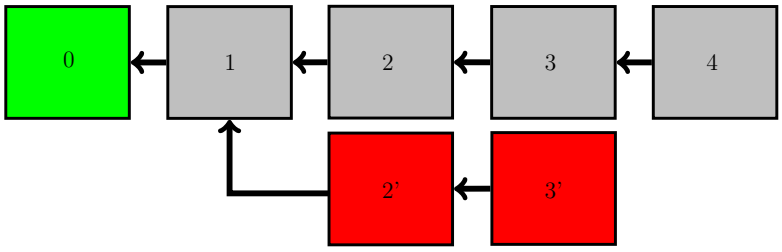

Figure 2.1: Blockchain formation. The gray boxes are blocks in the main chain. The green box is the genesis block. The two successors of the second block form a fork. The red blocks are orphans.

The first block of the chain has no predecessor, and is therefore called the *genesis block*. At any point there may be more than one block linking to the same predecessor, this is commonly referred to as *forking*, and leaves the network with an inconsistent view of the data. Due to the cryptographic link between the blocks, a fork will never be resolved by the merging of blocks, thus the most common strategy for handling forks is to only consider the longest chain of blocks to be valid and part of the main chain. The blocks outside the main chain are called *orphans*. Figure 2.1 illustrates a blockchain formation.

Due to the immutable property [33], by including a timestamp in every block, a blockchain can also provide an approximate idea of when each block was created [1], in addition to the exact order of creation [61]. The idea to link issued timestamps was first proposed in [75], and is known as *linked timestamping* in the literature [2].

A powerful usage of blockchains is as a distributed ledger [4, 12]. A distributed ledger can be thought of as a distributed database, in which the data is replicated, synchronized and shared between multiple participants over a peer-to-peer network. Each participant has their own identical local copy of the ledger, thus it is necessary to deploy consensus algorithms to ensure correct replication across all participants. Blockchains replicate that data only for resilience, not for scalability [4].

A distributed ledger reduces the systemic risk in comparison to centralized solutions [5]. However, the endpoint-security is far worse, as the node is responsible for safely and securely storing their private keys since there is no way to recover them if they are lost or stolen.

A blockchain system can also be classified as *atomic broadcast* or *total order broadcast*. Atomic broadcast ensures agreement on a common global order of messages amongst all the nodes in a system [10]. The analog for message in a blockchain is a *transaction*. Every block contains a batch of transactions. By batching the transactions, the number of communication steps to achieve global ordering is reduced, as opposed to agreeing on every message individually.

Many blockchain system allow the incorporation of *smart contracts* into transactions. These smart contracts are instructions which enforce the execution of arbitrary tasks. The smart contract may be written in a domain-specific or a general-purpose programming language [4], depending on its implementation. Blockchain systems have vastly different properties depending on their implementation. A comparison of a collection of existing implementations is given in Chapter 3.

Two main schools of blockchains exist: permissionless and permissioned. These will be discussed in detail in the following sections.

2.2.1 Permissionless Blockchain

Also known as a public blockchain, perhaps the most well-known example of a permissionless blockchain is Bitcoin. In principle, anyone with the appropriate software can become a participating node in the network. Due to being open to the public, no trust between the participants is assumed. Thus, the consensus protocol must be resistant to both naturally occurring faults, as well as maliciously crafted behaviors. Such a resistant consensus protocol is known in the literature as Byzantine fault tolerant, and is further discussed in Section 2.1.2.

A consequence of allowing anyone to create an unlimited amount of identities in the network, is the possibility of Sybil attacks [62]. A Sybil attack in this context means that an adversary forges enough identities to control consensus in the network. This is possible due to the fact that most fault-tolerant systems are reliant on some kind of majority of nodes are honest and reliable in order to make progress and achieve consensus [5]. In order to deflect Sybil attacks, permissionless blockchains use different functions to control *leader elections*, so that controlling a majority of nodes in the network will not be sufficient to control consensus. Examples of such functions are proof-of-work, proof-of-elapsed-time and proof-of-stake.

Proof-of-work was originally created as means to deter spam [5]. By re-

quiring the sender to include a solution to a puzzle which demand a moderate amount of computational work, it would slow down or deter a spammer.

In Bitcoin, proof-of-work is incorporated as a security mechanism limiting the creation of blocks. In particular, the requirement to append a block to the chain is that the hash value of the block must be under a given threshold. This is more commonly known as mining. Finding a hash value under a given threshold is often referred to as *difficulty* in the literature. Nodes are encouraged to find new blocks because whoever creates them will receive substantial monetary rewards in form of bitcoins. This results in an interesting power triangle consisting of the ledger rewarding miners, the miners that secure integrity of the ledger, and the perceived value of the currency rewarded. Interestingly, the transaction rate heavily impacts the security from double-spend attacks [53, 54]. The two parameters which limits the rate are *block size* and *block interval*. In Bitcoin, currently the average rate between blocks is 10 minutes, the block size 1MB [53, 3].

As everyone always could use another dollar, the monetization of such protocols leads to relentless competition. As noted in Section 3.5, Bitcoins total energy consumption is steadily increasing, and already surpasses entire nations. Thus, alternative approaches which are not dependent on immense computational power have been suggested. One such approach is proof-of-elapsed-time, which use secure enclaves in the CPU to control leader election. The secure enclave executes a waiting step in such a way that the other nodes can verify that the wait was executed properly. An implementation of proof-of-elapsed-time can be found in Hyperledger Sawtooth [50, 51], which builds on Intel's Software Guard Extensions (SGX) to generate the random delay. An obvious drawback of proof-of-elapsed-time is its reliance on custom hardware, especially if it's a single vendor that can produce it.

Ethereum is planning to move from proof-of-work to proof-of-stake using the Casper algorithm [52]. In PoS, each nodes chance to be able to propose the next block is proportionate to their stake in the system. If there are no incentives for a node to act honest, a node may vote on multiple blocks supporting multiple forks in an attempt to maximize their earnings [12]. This is also known as the *nothing-at-stake* attack, which Ethereum aims to counteract by penalizing malfeasant nodes [52].

2.2.2 Permissioned Blockchain

A permissioned blockchain, also known as a private or consortium blockchain in the literature, allows membership to be controlled by a system operator. Typically the admission into the network is controlled by a digital certificate. In most cases this will be issued by a certificate authority(CA), which is a trusted central authority. Analogous to other database systems, at a high level, a participant may take on one of two roles. The first being as a *writer*, also known as *validator*, which participates in the consensus protocol, and grows the blockchain. The other is a *reader*, also known as *non-validator*, which may only read and audit the blockchain. However, the blockchain system may have many more granular roles, such as writers limited to only certain type of transactions, or readers with only a partial view of the blockchain.

Under the assumption that the membership view is not compromised by an adversary, there is no risk of a Sybil attack [62]. With the absence of Sybil attacks, consensus protocols which are Byzantine fault tolerant can be used. BFT is discussed further in Section 2.1.2. These have been researched for decades [11], and for blockchain systems offer vastly different properties than their proof-of-work inspired counterpart [8, 12, 50]. In particular, BFT style consensus can offer vastly superior throughput and latency compared to proof-of-work. The reasoning being that these properties are coupled with security in proof-of-work. This is further elaborated on in Section 2.2.1. Another advantage with BFT style consensus is the immediate transaction finality, which means that once a block is added to the ledger, it will never be removed or otherwise considered invalid. Proof-of-work only offers probabilistic transaction finality, which means that there is a chance that multiple blocks will be added at the same height and cause forking. This means that a client will have to wait a long time until transactions are finalized and confirmed, and in Bitcoins case, up to 1 hour [1, 2]. There is also a difference in the form of adversary tolerance, where BFT consensus can tolerate less than $\frac{1}{3}$ of the validators being Byzantine, and proof-of-work less than $\frac{1}{4}$ of the computing power being controlled by adversaries [53].

Even though it is possible to design a permissioned blockchain without BFT consensus, to address research question **RQ3**, a major drawback of BFT consensus is that scalability of the number of validators is limited. At a certain point the amount of overhead in the messages exchanged between nodes becomes crippling. Some recommend that the number of validators be

kept no higher than 20 [53, 12]. Chapter 3 contains a comparison of various implementations of permissioned blockchains.

2.3 Cryptographic Hash Functions

The job of a basic hash function is to map an arbitrary long message into a fixed length value. In order for the hash function to be classified as a cryptographic hash function, and be useful in blockchain systems, several key properties are required [9];

1. **Preimage resistant (Hiding):** Given any output value $y = \mathrm{H}(x)$, it is computationally infeasible to find x.

2. **Second preimage (Puzzle-friendliness):** Given any input value x, finding a different input value y such that $\mathrm{H}(x) = \mathrm{H}(y)$ is computationally infeasible. Meaning that it is very difficult to target the output of the hash function to a particular value.

3. **Collision resistant:** Finding a pair of two unique input values that map to the same output value is computationally infeasible.

4. **Efficiency:** It is computationally cheap to calculate a hash value.

5. **Pseudorandomness:** The output hash values will be distributed in such a way that it will pass standard pseudorandomness tests.

The term preimage means that for a hash value $h = \mathrm{H}(x)$, x is the preimage of h. The definition for the hiding and puzzle-friendliness properties given here differs slightly from [1], as we have omitted the secret random value concatenated with the input value.

Vulnerabilities on the hashing algorithms are constantly researched. In addition to this, the power of supercomputers is ever-increasing. Both these factors force the constant development of new and stronger cryptographic hash functions. A notable example of a hashing algorithm which previously was considered cryptographically secure, and is now broken, is MD5. MD5 was first published in 1992 [19], and by 2012 it had been used as an attack vector several times, most infamously by the Flame malware [20] in which a collision attack was performed to forge a certificate to sign code on Windows systems.

One of the most widely used cryptographic hash functions are those in the SHA-2 family. Applications include SSH, TLS, SSL, several blockchain systems and many others. In particular the SHA-256 algorithm is used in Bitcoin and other cryptocurrencies for calculating proof-of-work and verifying transactions. In 2015 the latest member of the SHA family, SHA-3 was released [21].

2.4 Digital Signatures

A digital signature is a scheme built on public-key cryptographic used to provide data authentication between two parties which is verifiable by third parties. In most blockchain systems, digital signatures are an essential component as they allow all the readers to verify the originator of every transaction and block.

Before an entity is able to digitally sign, two distinct keys must be generated; a public key and a private key [10]. This is known as public-key cryptography or asymmetric cryptography in the literature. The keys need to be generated in a secure way, and it is especially important to have a good source of randomness.

The public key is used to verify your signature, and is assumed to be known by everyone in the network. In fact, some systems will use the public key as your pseudonym or address. On the other hand, the private key will be used to sign, and thus it is of utmost importance to keep it secure, as a compromised private key can be used by an attacker to sign on your behalf.

The desired security properties of a digital signature are described in [9];

Authentication High confidence in the identity of the communicating entity.

Data integrity Assurance that the received data has not been tampered with by an adversary.

Non-repudiation It is not possible to deny having sent data which the user has signed.

These properties are obtained through the following design requirements of a digital signature scheme [9];

- The signature must depend on both the data being signed, and the user signing it. This property is crucial for providing non-repudiation and preventing forgery.
- It is computationally cheap to both produce the digital signature and for another party to recognize and verify it.
- It is computationally infeasible to forge a signature without knowing the private key.

Public keys may be distributed in a number of different ways, as the security properties assume that the public key is known by everyone. The main challenge in distributing the key lies in proving the link between your real-life identity and the key.

2.5 Message-Authentication Codes (MACs)

As an alternative to digital signatures, a *message-authentication code (MAC)* can be used to authenticate data exchange between two entities [10]. The MAC function may be built using symmetric encryption, such as cryptographic hash functions or block ciphers [9].

Due to them not being reliant on the computationally expensive asymmetric ciphers, MACs are computationally cheaper to compute and verify compared to digital signatures [10]. The drawback is that MACs do not provide the non-repudiation property, meaning that there is no way to prove who sent a message.

2.6 Merkle Trees

A Merkle tree [55] is a tree-based datastructure used for efficient integrity verification. Its leaf nodes are computed by hashing the input data blocks, and all other nodes are computed by hashing its child nodes. Most commonly in blockchain systems each internal node will have no more than two children, making it a binary tree. An example of a Merkle tree is shown in Figure 2.2. A typical usage in blockchain systems is having a every transaction contained within a block serve as input to the tree, and for the root hash to be included in the block header [30, 1].

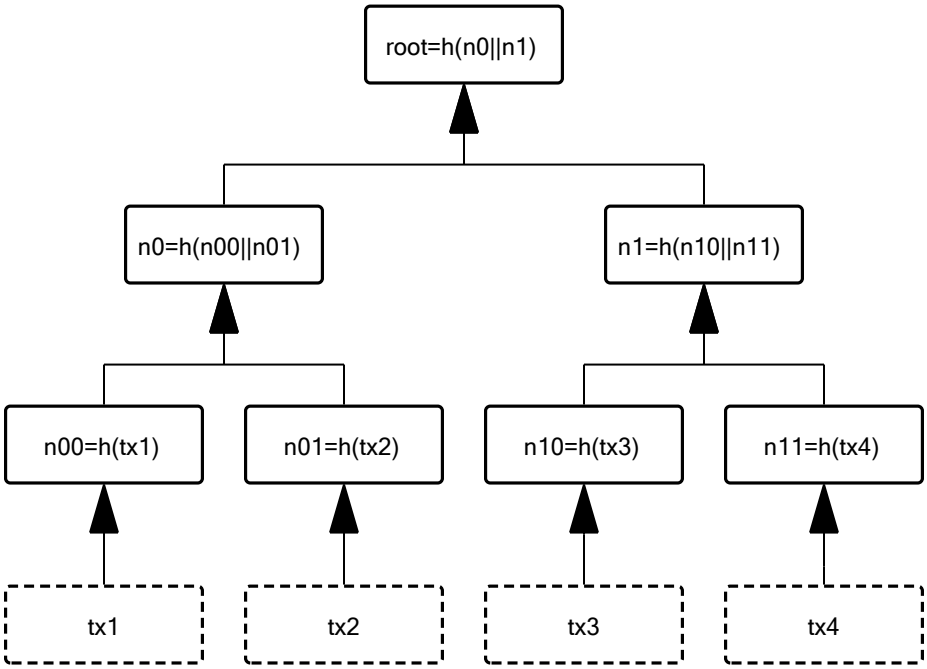

Figure 2.2: Binary Merkle tree. The value of leaf nodes is the hash of a data block, and the value of every non-leaf node is the hash of its children.

The integrity verification is efficient due to it only being necessary to look at the value of the root node to detect modifications. The reason being that any change to a node or the tree structure will propagate all the way up to the root, and thus change its value.

Another important property of a sorted binary Merkle tree is that of efficiently proving whether or not a particular leaf node is part of the tree. Since we only need to compute the hashes along the path from the root to the leaf, the number of hashes required to compute are logarithmically proportional to the number of nodes in the tree [1].

2.7 Peer-to-Peer Networks

A cornerstone to achieving true decentralization, is having peers operate on equal terms. The peer-to-peer (P2P) network describes an architecture where there is no centralized control or hierarchical organization, and every node runs software with equivalent functionality [24]. All the nodes participating are considered equals and share the responsibility of keeping the network operational [3]. The P2P networks are usually implemented on top of the existing network topology, where it operates as an overlay network [29] to facilitate the logical direct communication between nodes [25].

P2P networks are used in a wide range of applications [29, 25, 2], such as file-sharing, data distribution, Internet telephony, scientific computing, secure Internet communication and blockchains. Examples of P2P systems used for file-sharing include Bittorrent and IPFS. IPFS will be discussed further in Section 3.1. These systems can scale to very large networks, and use a Distributed hash table [25, 28] to map file-identifies to the peers sharing the files, enabling lookup in $O(log\ n)$ time.

In order for a node to disseminate information to all other nodes, it may use a *broadcast* abstraction. To ensure reliable message delivery in the presence of faulty processes or links, the node sending the message also needs to receive some form of acknowledgment [10]. This approach does not scale well, as it requires $O(n^2)$ messages, and may therefore become a severe bottleneck in a system with many participants. Instead, a probabilistic broadcast approach such as the gossip protocol provides a scalable option. The following description of the basic operations of a gossip protocol is adopted from [10].

The node wanting to broadcast a message will select k nodes at random, and send them the message. Then every node that received the message selects another k nodes at random and forward the message to them, and so on. This will continue for R rounds, and thus the probability of every node in the network receiving the message is dependent on the *fanout* parameter k and number of rounds R. Due to the probabilistic nature of the protocol, reliable broadcast is not guaranteed, and thus it may terminate before every node in the network has received the message.

However, using gossip protocol in a State machine replication setting with Byzantine fault tolerance systems adds another level of complexity, as replicas might be reluctant to update their state without receiving confirmation from a quorum of correct nodes. In particular, vulnerabilities of a gossip-based protocol include Denial-of-service attacks targeted at a small subset of the

correct nodes [27], and network partitioning in which the nodes in either partition might enter different states. Examples of BFT systems using gossip-based broadcast include Fireflies [25], which is a overlay network protocol that provides a secure and scalable membership service. Fireflies organizes the nodes in a pseudorandom structure, in such a way that every node is monitored by at least one correct node.

Most blockchain systems use gossiping to disseminate new blocks to all nodes in the network. This is true for both schools of blockchains. In the case of permissionless blockchains using proof-of-work consensus, such as Bitcoin and Ethereum [1], it is well understood that the propagation of new blocks might be delayed, therefore miners may create an relay network between themselves to minimize the latency of transmission of new blocks [3], as after all, *time is money*.

2.8 Proof of Storage

The most simple way of proving to someone that you possess some piece of data is to send them the data in its entirety. However, this is problematic both in terms of network bandwidth, which is a limited resource, and hard-drive I/O since the performance of magnetic disks has not improved as rapidly as its capacity [45]. For this reason, it must be possible able to provide non-reputable proofs without accessing the entire file.

The basis of the proof is to have the verifier **V** store a constant amount of metadata which is kept secret from the prover **P** and used to generate probabilistic storage challenges **C**. Examples of such algorithms are [14, 17, 18]. In particular [17, 18] computes an *homomorphic authenticator* for every file block, and stores this alongside the file at **P**. The homomorphic authenticators acts as verification metadata, and using them **P** is able to combine blocks and authenticators into a single aggregate block and authenticator [18]. A challenge-proof cycle may be summarized as follows:

1. Verifier **V** generates a random challenge **C** and sends it to **P**.
2. **P** computes the proof based on **C** and sends it to **V**.
3. **V** verifies the proof. If the proof does not match, **P** node will be marked as faulty. Failure to respond will also result in being marked as faulty.

The probabilistic proofs are created in such a way that false negatives may not occur, only false positives. Meaning that, even though a proof matches, **P** may still be faulty, but the challenge was not strong enough to detect it.

2.8.1 Public Verifiability

Not to be confused with the property which allows verification of data integrity in a blockchain, in the context of proof-of-storage the *public verifiability* property refers to the ability of anyone, not just the data owner to take on the role as the verifier. The algorithms in [14, 17, 18] can all be modified to offer this property.

Chapter 3

Related Work

This chapter starts off with Table 3, showing an overview of the characteristics of different consensus algorithms in blockchain systems. The table uses data found in [50, 12, 8, 15]. After the table, the two systems our prototype builds upon, IPFS and Tendermint will be introduced. Then we will take a look at two of the blockchains in the Hyperledger framework, then Corda, a system which is inspired by blockchains and then Bitcoin. Finally three systems which resembles ours, Filecoin, Sia and Storj will be described.

	BFT	**PoW**	**PoS**	**PoET**
Blockchain type	Permissioned	Permissionless	Both	Both
Throughput	High	Low	Limited	Limited
Finality (Latency)	Immediate	Probabilistic	Probabilistic	Probabilistic
Number of readers	High	High	High	High
Number of writers	Low	High	High	High
Maximum adversaries	Up to 1/3	Up to 1/4	Depends on algorithm	Unknown
Tokens required?	No	Yes	Yes	No

Table 3.1: Comparison of consensus algorithms used in blockchain systems.

3.1 IPFS

IPFS is a content-addressable, distributed P2P file system [28]. IPFS incorporates ideas from BitTorrent, Git, DHT and SFS. All the stored data is content-addressable (CAS) by its unique, immutable content identifier, also abbreviated as *CID*. In addition the fact that the CID is generated using a Merkle DAG, means that any changes to the data can be easily detected.

By default IPFS comes with a set of bootstrap nodes that it will attempt to form a network with. However, in order to form a private network, IPFS-Cluster may be used in a layer on top of IPFS. IPFS-Cluster facilitates the dissemination of stored data in order to avoid a single point of failure. The leader-based consensus algorithm *Raft* [38] is used to coordinate the state amongst the IPFS nodes. Raft is crash-recover for benign faults, which means nodes are assumed to fail by stopping or crashing. Malicious behavior, such as sending conflicting to the network is not supported. However, as elaborated on in Chapter 4, our design relies on proof-of-storage to detect malicious behavior of the storage nodes. Once a cluster is up, peers are expected to run continuously [32].

The distributed log of Raft which every node follows is appended using mainly two operations; *Pin* and *Unpin* [32]. The Pin-operation tells a node to store a piece of data locally, and will thus disseminate the data throughout the network. The Unpin-operation is the opposite, as it tells the nodes that it is no longer necessary to keep some piece of data. Note that Unpin is not an explicit call to delete the data, as it is up to the individual IPFS node to decide when it should be permanently deleted. The cluster can not enforce forgetfulness guarantees regarding the deletion of data, meaning that a malicious individual node that refuses to delete some data will be undetectable.

3.2 Tendermint

Tendermint is a set of software used for BFT state machine replication in distributed applications [31]. It consists of two main components; a blockchain consensus engine called *Tendermint Core*, and a generic application interface. Our implementation is mainly concerned with the generic application inferface, also referred to as ABCI, but a description of Tendermint Core will be given here for completeness.

Tendermint Core uses a permissioned blockchain, where only actors with a registered public-/private-key pair may be able to participate in the consensus protocol. The consensus protocol was originally based on DLS [36], but has since been resigined to resemble PBFT [34, 33]. Participants in the consensus protocol are split into *validators* and *non-validators*. The difference being that a non-validator does not sign any votes, but they still keep up with consensus and process transactions. The protocol consists of three-step rounds, where a designated validator will be deterministically voted to act as the proposer for a single round. The proposer will broadcast a set of transactions to be included into the next block in the chain. *Safety* and *liveness* are guaranteed as long as less than $\frac{1}{3}$ of the validators are faulty.

Algorithm 1 gives a simplified view of the Tendermint Core. The protocol uses the partly synchronous model. In particular, the validators will only wait a small amount of time for a new proposed block before voting a new validator to act as the proposer in a new round. The proof-of-stake algorithm of Tendermint allows easy manipulation of voting power amongst the validators. A quorum in the consensus protocol is actually based on this voting power, and not the number of nodes participating.

The protocol starts once a validator receives a transaction from a client. The transaction is validated by calling the applications *CheckTx* method. The details of the implementation of CheckTx are discussed in Section 5.4. If CheckTx returns a *OK* status code, the transaction will be added to a memory cache and relayed to the other validators, any other status code results in the immediate discarding of the transaction. The memory cache of transactions is more commonly known in the literature as *mempool*.

After the proposer broadcasts a new block for inclusion in the chain, two phases of voting will take place amongst the validators. The first phase is called *pre-vote* and the latter *pre-commit*. Both phases require that more than $\frac{2}{3}$ votes for the same block in the same round to make progress [31]. The transactions in the block will be ordered. After a successful pre-commit phase, a *commit* is executed, and the new block is appended to the tail of the blockchain.

Once a new block is committed, Tendermint Core will attempt to execute the associated applications *DeliverTx* method once for every transaction in the block. This can be used to do any state transitions associated with a single transaction. Though not necessary for our prototype, it is important to note that since the transactions are ordered, every node will execute DeliverTx in the same order. Additionally, DeliverTx may return the result of

its execution, making it possible to report faults in the actual transaction. It is left up to the application developer to make sure that the execution is deterministic, so that we end up with a consistent state amongst the replicas. Section 5.4 discusses determinism further. DeliverTx is sandwiched by two other calls to the application stack, *BeginBlock* and *EndBlock*. Though not used in the prototype, these may be used to persistently store the application state or to change the validator configuration for the next block. After EndBlock, a final call to *Commit* is done to compute a cryptographic commitment for inclusion into the next blocks header.

All of the transactions contained in the block are then removed from the mempool, and the remaining transactions are then re-evaluated against CheckTx. The reasoning being that these transactions may no longer be valid as the new block could have changed the applications state.

Algorithm: 1 Ledger node (Tendermint Core)

Uses:
Mempool, **instance** *mem*.
Application, **instance** *app*.
EpidemicGossip, **instance** *ego*.

upon event $\langle cor, Init \rangle$ **do**
proposalWait := *0*; // Monotonically increasing every second
uniqueid := *0*;
height := *0*;
round := *0*;
votedBlock := *nil*;
mempool := $\emptyset$;
prevotes := $\emptyset$;
precommits := $\emptyset$;

upon exists $tx \in mempool$ **do**
if proposer(self, round) **then**
block := $\emptyset$;
forall $tx \in mempool$ **such that** |block| $< \alpha$ **do**
block := *block* $\cup$ {*tx.tx, tx.uniqueid*};
trigger $\langle Broadcast \mid [\textsc{ProposalMsg}, block, h, r] \rangle$;

upon exists *proposalWait* = *ProposalTimeout* **do**
  // Weak synchrony assumption
  trigger ⟨*Broadcast* | [VoteMsg, *nil, h, r*]⟩;

upon event ⟨*cor, NewTx* | *q,* [ProposalMsg, *block, h, r*]⟩ **do**
  if *h+1* > *height* **then**
    trigger ⟨*ego, Gossip* | [LaggingBlock, *h*]⟩;
    return
  else if *h* ≤ *height* ∨ *round* ≠ *r* **then**
    return
  forall *tx* ∈ *block* **do**
    if *app.CheckTx(tx)* = False **then**;
      trigger ⟨*Broadcast* | [VoteMsg, *nil, h, r*]⟩;
      return
  votedBlock = *block*;
  trigger ⟨*Broadcast* | [VoteMsg, *block, h, r*]⟩;

upon event ⟨*cor, NewTx* | *tx*⟩ **do**
  if *app.CheckTx(tx)* = True **then**;
    mempool := *mempool* ∪ {*tx, uniqueid*};
    uniqueid := *uniqueid* + 1;

upon event ⟨*cor, Prevote* | *q,* [VoteMsg, *block, h, r*]⟩ **do**
  prevotes := *prevotes* ∪ {*block, h, r*};

upon exists |*prevotes*| > $\frac{2n}{3}$ **such that** ∀ {block, h, r}:
      block = votedBlock, h = height, r = round **do**
  prevotes := ∅;
  trigger ⟨*Broadcast* | [Commit, *block, h, r*]⟩;

upon event ⟨*cor, Precommit* | *q,* [Commit, *block, h, r*]⟩ **do**
  precommits := *precommits* ∪ {*block, h, r*};

upon exists |*precommits*| > $\frac{2n}{3}$ **such that** ∀ {block, h, r}:
      block = votedBlock, h = height, r = round **do**
  precommits := ∅;
  if *block* = *nil* **then**;
    round := *round* + 1;

```
    else
        forall tx ∈ block do
            app.DeliverTx(tx);
        Commit();

procedure Commit
    height := height + 1;
    round := 0;
    mempool := mempool \ votedBlock;
    votedBlock := nil;

upon event ⟨cor, Query | [path, DATA]⟩ do
    trigger ⟨cor, QueryResponse | node, [app.Query(path, DATA)]⟩;
```

3.3 Hyperledger Frameworks

Hyperledger is a collection of multiple blockchain frameworks hosted by the Linux Foundation. The frameworks seems to be specialized towards business applications [50]. We will take a look at two of the frameworks; *Fabric* and *Sawtooth*.

3.3.1 Hyperledger Fabric

Hyperledger Fabric is a platform for distributed ledger solutions [37]. It is written in Golang with a modular architecture so that components such as consensus and membership services are plug-and-play. Its goals are to provide high degrees of confidentiality, resiliency, flexibility and scalability [37].

The type of blockchain is permissioned, meaning that only known entities can participate in the network. Consensus is broken into three phases [50]; Endorsement, Ordering and Validation. Hyperledger Fabric supports pluggable consensus for all the phases, giving the application developer great flexibility.

For the ordering service, Hyperledger Fabric provides Apache Kafka as the consensus mechanism [4, 50]. In fact, Hyperledger Fabric assumes expertise

with Kafka [37] to use it in an ordering service. Apache Kafka [65] is a distributed streaming platform aimed at handling large amounts of real-time data. It uses a publish-subscribe interface to send messages, allowing multiple readers for each message. Kafka uses Apache ZooKeeper to coordinate its brokers internally [37], and tolerates up to $f < \frac{n}{2}$ crash faults [65]. Fabric inherits this resilience [4].

3.3.2 Hyperledger Sawtooth

Sawtooth is a permissionless blockchain for general-purpose smart contracts [4]. It uses a lottery-based consensus algorithm called proof-of-elapsed-time [50]. As with proof-of-work, there has to be a waiting period between leader elections. Originally contributed by Intel [4], proof-of-elapsed-time aims to execute this waiting period without expending huge amounts of energy. The protocol is reliant on a secure enclave on the CPU in order to provide proofs for executing the waiting period.

The secure enclave executes a waiting step in such a way that the other nodes can verify that the wait was executed properly. The secure enclave can be found on Intel's Software Guard Extensions (SGX). It relies on custom hardware, and the probability of winning the election is proportional to the number of nodes in the network.

3.4 Corda

Corda is an open-source distributed ledger platform developed by R3 [7], designed for semi-private networks in which admission requires obtaining an identity signed by a root authority [6]. It has a special emphasis on the financial world and business applications. Though it shares many of the benefits of a blockchain, it differs from the definition given in Section 2.2. In particular, the transaction execution model produces a hashed directed acyclic graph [4]. There is no broadcast of transactions in the network. All communcation is directed, and thus data is shared on a need-to-know basis [6]. This gives nodes in the network partial visibility of the entire transaction graph. Different organizations may merge their ledgers by establishing a two-way trust between *notaries* and certificate authorities.

The Corda network does not organize time into blocks [6], instead it uses notary services to provide transaction ordering and timestamping. Each

network may consist of multiple notaries, and they may each run different consensus algorithms. The fault tolerance of the network depends on the choice of the consensus algorithm. Corda offers support for running Raft or BFT-SMaRt [4] on notary nodes. Raft gives tolerance to crash-faults, and can tolerate up to $f < \frac{n}{2}$ faults. BFT-SMaRt [63, 64] tolerates up to $f < \frac{n}{3}$ Byzantine faults.

3.5 Bitcoin

The global phenomena, Bitcoin [61], is a decentralized cryptocurrency, and first of its kind [5]. Bitcoin runs on a permissionless blockchain, allowing anyone to join the network. The consensus mechanism is proof-of-work. To handle forking, the network will consider the longest chain to be the valid chain. This is analogous to the one with the most accumulated computational effort [73].

Every block in the chain contains a bundle of transactions. Every transaction specifies a number of inputs, a number of outputs, and some metadata [1]. In effect it moves coins from one entity to another. Due to fact that anyone can join the network, each entity is only associated with a public key, and not a real life identity. All the inputs of the transaction have to be digitally signed for the transaction to be considered valid. The recipients of the coins do not have to be online or take any action for the transaction to be valid. In addition, it is possible to use the outputs to specify more advanced scripts. Bitcoin script are written in its own stack-based language [3], which offers only a limited set of instructions. The scripts have to be deterministic, and can have only one of two outcomes after executing, *success* or *error*.

On average, a new block will be mined every 10 minutes [3]. In the case of Bitcoin, mining a new block means to find a nonce so that the hash value of the proposed block will be under a given threshold. The protocol self-adjusts this threshold in such a way that it's proportional to the amount of computational power available in the network.

Several studies [60, 53, 26] have suggested to either increase the block size or reduce the block latency as means to improve performance. However, adjusting these parameters have consequences for the consensus mechanism. In particular, with a larger block size it will take longer for it to propagate in the network, and a lower block latency leads to increased forking and instability when agreeing on the valid chain [53]. Currently the throughput

of Bitcoin is limited to 7 tx/sec [3, 1], several orders of magnitude lower than Visa, which processes 2,000 tx/sec on average [2], with peaks of up to 59,000 tx/sec [60].

A major drawback of Bitcoin and proof-of-work in general is that the consensus mechanism consumes enormous amounts of energy [12, 30]. Bitcoin is vulnerable to selfish miners controlling more than 1/4 of the computational power in the network [53]. A selfish miner will withhold blocks it has mined from the network for as long as possible in order to get an unfair advantage mining the next block in the chain.

3.6 Filecoin

Filecoin [13] is a decentralized storage network that serves as an incentive layer on top of IPFS [28]. The protocol uses its own token, *filecoin* as the currency to pay for services. The network is comprised of clients who pay for storing and retrieving data, storage miners that earn tokens by storing data, and retrieval miners that earn tokens by serving data.

To organize the interactions between the client and the miners, a permissionless blockchain is used, also referred to as a ledger. The ledger is made up of transactions specifying storage- and retrieval orders from the clients, storage pledges from miners and proofs of storage. In addition, each block has an allocation table which keeps the state of network by specifying on which storage miner every piece of data is stored.

Filecoin introduces two novel proof-of-storage algorithms. The first is proof-of-replication which is used to prove that the data has been replicated to its own uniquely dedicated storage. The second is proof-of-spacetime which aims to verify that the data was stored for a specific time interval.

As opposed to the wasteful proof-of-work algorithm of Bitcoin, Filecoin aims to let the work done by miners be useful. In particular, the probability to be elected as the next block creator is proportional to amount of data stored for the network [13].

3.7 Sia

Sia [66] is a decentralized open-source cloud storage platform built on a permissionless blockchain. The system allows peers on the network to rent

storage from each other. The currency used to pay for services in Sia is called *siacoin*. Siacoin may also be obtained through mining or trading [66]. Only the storage contracts defining the terms of the agreement are stored on the blockchain. The payload data is stored on the storage providers, also known as hosts.

Similar to Filecoin, a host agrees to store a client's data and to periodically submit proof that the data is stored while the storage contract is valid. Clients may protect themselves from hosts going offline and thus their data becoming unavailable by striping the file into chunks, and then applying erasure codes to split each chunk into multiple pieces, but only requiring a subset of pieces to reconstruct the file. Currently, each chunk of the file is split into 30 pieces, while requiring 10 pieces to reconstruct [67]. Sia encrypts each piece using the Twofish algorithm [67].

All the pieces will be disseminated to a total of 50 hosts, of which no host may hold multiple pieces for a single chuck and the target redundancy for a piece is 3 times [67]. Each host must have 97% uptime in order to fulfill the storage contract. Additionally, the host must regularly submit storage proofs to the network. While Filecoin and Storj have a maximum token supply, siacoin is monotonically increasing [66].

3.8 Storj

Storj [71] is an open-source peer-to-peer cloud storage network built on the distributed hash table (DHT), Kademlia [70]. Storj uses a permissionless blockchain to manage the metadata of the stored data [72]. Similar to Filecoin [13] and Sia [66], the blockchain is used to detect unauthorized modification and deletion of data. In Storj, only the data owner can issue storage challenges. However, the proofs are recorded on the ledger and thus anyone can verify them.

The currency used to pay for services is called *storjcoin*. To improve performance and availability, every uploaded file goes through a sharding process. This means that every file is split up into multiple smaller pieces, which are then scattered to different storage hosts on the network. In addition, this allows faster retrieval as one can download the shards in parallel [71]. Clients are expected to encrypt the data before uploading, and are themselves responsible for keeping the encryption key secure.

Chapter 4

Charting the Design Space for Ledger-based Storage

In this chapter, we outline the principles and design ideas for the thesis, whose overall aim is to design a highly scalable architecture for a storage system built on blockchain technology that requires strong data integrity. The stored data should be permanently available, and its authenticity must be verifiable. The system should be able to support concurrent reading and writing by clients.

As already mentioned in Chapter 3, we have identified Filecoin [13], Sia [66] and Storj [71] as similar systems. However, they all use a permissionless blockchain, allowing nodes to act as miners by storing data and providing bandwidth. The miners are incentivized by rewarding them with the systems own currency, respectively *filecoin*, *siacoin* and *storjcoin*. The fact that anyone can join their network and store sensitive data forces the systems to implement several mechanisms to ensure the permanent availability of the data. In particular, churn rate might be high, resulting in the need of a high redundancy factor to ensure the availability of the data and keep the system running. The protocol also need to be resistant against a malicious entity creating several identities in the system in an attempt to gain monopoly over some piece of data. Even if the data was encrypted, if the malicious entity held the only copy of some data it could attempt to extort the owner of the data. In addition, the clients will periodically ask the storage nodes to provide proof-of-storage, which will consume a lot of bandwidth as the redundancy factor and the number of hosts increases.

We design our storage system with a permissioned blockchain, and adopt

a hybrid decentralized model where the ledger- and storage nodes are operated by trusted system operators. The use of permissioned blockchains allows us to use traditional BFT consensus, which enables the system to provide low latency for client requests and high throughput for transaction processing.

Throughout this chapter we will attempt to link our findings to the research questions posted in Chapter 1.

4.1 System Model and Assumptions

The system assumes the partially synchronous model [10, 30], there is no agreement on global time, and the computation speed of every operation and network delay is assumed to be bounded with an unknown upper bound Δ relative to real time. The nodes are connected in a network where the links are unreliable, and that may fail to deliver the messages, reorder them or duplicate them. The nodes have access to an internal clock which has a bounded difference to the global clock, and there is a finite latency on message exchange between correct nodes.

The system uses a Byzantine failure model, in which faulty nodes may behave arbitrarily, and even maliciously. Faulty nodes do not possess the computing power needed to circumvent the cryptographic primitives, and there are no known methods to trivialize their security mechanisms. Correct nodes never divulge their private keys to any other node, and a node cannot steal a private key by any means.

The system can guarantee liveness and safety under the following conditions; $n_c \geq f_c$, $n_l > 3f_l$, $n_s > 2f_s$. Where n_c, n_l, n_s means the total number of clients, ledger nodes and storage nodes, respectively, and f_c, f_l, f_s means the number of Byzantine faulty clients, ledger nodes and storage nodes, respectively.

We assume that the ledger- and storage nodes possess a full membership view of the system. The clients have a partial membership view, as they need only know of the ledger- and storage nodes. Thus, a client may multicast its messages to $f + 1$ different nodes in order to reach at least one correct node, assuming no network partitioning.

Nodes can verify the origin of any message they receive. In particular, all messages between components are digitally signed. The public and private key-pairs used for digital signatures are certified by a trusted CA. The rea-

son for using digital signatures as opposed to MAC is the non-repudiation property, and that digitally signed messages can be put on the ledger and be verified by everyone without exposing the signing key.

4.1.1 Attack Model

We assume the nodes are protected behind firewalls in order to thwart attacks such as intrusion and Denial-of-service. Insider attacks [76], such as an operator that compromises a quorum of the ledger nodes are not considered in this thesis, as this would be solved outside the protocol. We do not consider any countermeasures to prevent uploading illegal content.

As noted in [25], errors in the mechanisms that maintain the membership view can be detrimental to the protocol. In particular, an attacker may add or remove members in such a way that the stated fault tolerance is circumvented. As the membership view resides locally on each member, we assume that such an attack is not possible.

Incoming transactions will be kept in cache, also known as the mempool until their validity can be determined. Thus, it is vulnerable to denial of service attacks [33].

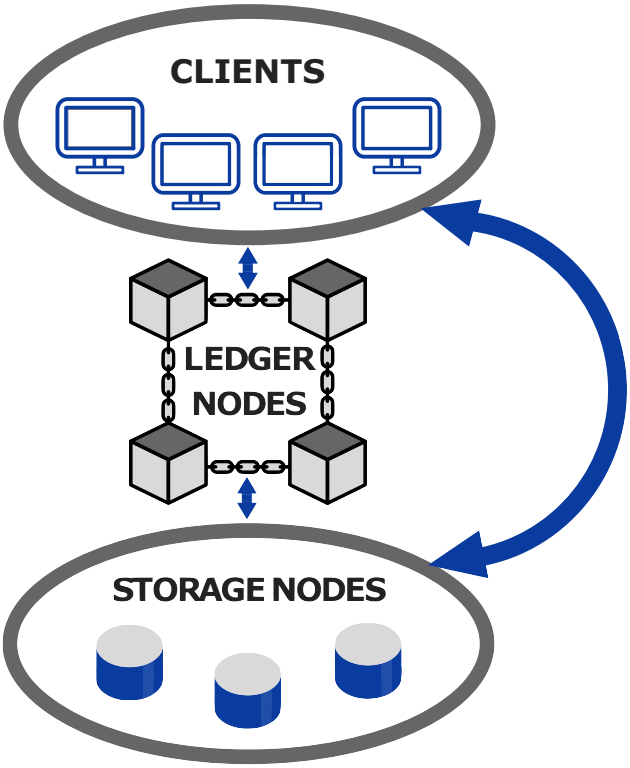

Figure 4.1: Architecture

4.2 System Architecture

The architecture designed is shown in Figure 4.1, and is organized into three core components; *clients*, *ledger nodes* and *storage nodes*. The storage nodes are responsible for keeping the stored data in persistent storage. The ledger nodes are responsible for maintaining the blockchain, which contains pointers to the stored data.

The storage nodes are connected in a peer-to-peer network, and each has a full replica of the system's stored data. A client is able to upload arbitrary data to a storage node, as well as download it. The storage node uses content-addressing [28] so that each piece of data is identified by its unique content identifier, *CID*. Once a new block is committed to the blockchain, each storage node receives the update. In addition, the storage nodes possesses the

ability to disseminate data amongst themselves.

By using a blockchain to keep track of the stored data, we gain properties such as public verifiability of data authenticity and linked timestamping [1]. In addition, by having the storage nodes publish non-reputable proof-of-storage to the blockchain, the data availability of the system can be publicly verified.

As mentioned in Chapter 3, our study reveals that permissioned blockchains allows the usage of traditional BFT algorithms, and thus can offer superior throughput and latency whilst providing strong data integrity, and therefore we opt to use it in our design. This also answers research question **RQ1**. The system requires $3f_l + 1$ ledger nodes to tolerate f_l Byzantine-faults amongst them. As noted in [43, 44], the separation of agreement and storage allows the reduction of the number of storage nodes to $2f_s + 1$ whilst tolerating up to f_s Byzantine-faults.

The public verifiability of data authenticity is provided by having each transaction be digitally signed, and the fingerprint of the public key, or pseudonym of the signer be included in the transaction. Thus, the privacy of clients is reliant on the linking of the pseudonym to their real life identity. As a permissioned blockchain requires membership to be controlled by a system operator, he may be able to perform this linking, depending on the registration policy. Therefore clients can not expect to remain anonymous. This is one of the challenges of permissioned blockchains, as asked in research question **RQ3**.

We design our system to be highly available, but with regards to the CAP theorem [46], availability is sacrificed for consistency in the event of network partitions. As a new block requires a supermajority of more than two thirds of the ledger nodes to validate, it is not possible to create two large enough partitions without having at least one correct node participate in both. Therefore *forks* may not exist, and there is no need for mechanisms to solve conflicts. With regards to storage nodes, they will allow a client to upload new data during partitions, but will not disseminate this data to the other nodes until the upload transaction is committed to the blockchain. Downloading data requires that only one correct node is reachable. In addition, high availability is achieved by having a large number of redundant ledger- and storage nodes which allows the masking of benign and Byzantine faults.

In terms of scalability, we design the architecture to be scalable both in storage capacity and throughput — without sacrificing latency or integrity.

The separation of those that handle agreements in the ledger nodes, and those that actually store the payload data in the storage nodes allows for great flexibility. In particular, it might be reasonable to have more ledger nodes, while not needing as many replicas to store the payload data. As the usage of the system increases, this separation allows the ledger nodes to scale vertically by adding more CPU and memory, and the storage nodes to scale horizontally by adding more nodes to increase storage capacity.

Strong data integrity gives clients confidence in the system, as they can be certain that all data modifications are detectable. Integrity is guaranteed through a two-pronged approach. First the integrity of the blockchain is preserved by having each block link back to its predecessor. Second, the integrity of the stored data is preserved by using immutable content addressing and a pointer contained within the block.

Even though the integrity guarantees of the stored data is preserved, any modification is not detected until the data is accessed. This is problematic because some data may very rarely be accessed. For this reason, the system provides public verifiability of data availability by giving clients the power to make storage nodes provide non-reputable proof-of-storage to be committed on the blockchain.

4.3 Specification of Storage Operations

The system is operated using five operations; *Change Access*, *Upload*, *Download*, *Delete*, and *Verify Storage*. All operation- and intermediate messages are digitally signed by the originator. The following sections starts off with an informal description of each operation, followed up with a slightly more formal description of the necessary pre- and post-conditions. More details about the implementation can be found in Section 5.1.

4.3.1 Change Access

The system defines two roles: *owner* and *reader*. The owner is always the one that originally uploaded the data. The owner can modify the list of readers. Note that the system can not enforce forgetfulness on users who has downloaded data previously, but has had their access revoked. The necessary pre- and post-conditions are defined as follows:

Pre Condition there is a transaction committed to the blockchain claiming the availability of some data element d identified by $\mathrm{H}(d)$ for which the client is identified as an *owner*, and all storage nodes store a data element d identified by $\mathrm{H}(d)$.

Post Condition new transaction claiming the availabiliy of $\mathrm{H}(d)$ with a modified list of readers is committed to the blockchain, and all storage nodes store a data element d identified by $\mathrm{H}(d)$.

4.3.2 Upload

This operation is used when a client wants to store data in the system. In order to reduce the impact on the clients bandwidth, it is only required to upload the data to a single storage node. This storage node will then disseminate the data to the rest of the storage nodes. The necessary pre- and post-conditions are defined as follows:

Pre Condition a client wanting to store data element d with an unique content identifier, $\mathrm{H}(d)$, and no data element d' with content identifier $\mathrm{H}(d)$ is stored on any storage node by the client.

Post Condition a transaction stating the availability of $\mathrm{H}(d)$ is committed to the blockchain, and data element d has been disseminated to all storage nodes.

4.3.3 Download

When a client wants to download a data element, it must be recognized as a reader for that particular element. This is the only operation which does change the systems state. Upon receiving the data element, the client is able to verify its authenticity by recalculating the content identifier and comparing it to the one published on the blockchain. The necessary pre- and post-conditions are defined as follows:

Pre Condition there is a transaction committed to the blockchain claiming the availability of some data element d identified by $\mathrm{H}(d)$ for which the client is identified as a *reader*, and all storage nodes store a data element d identified by $\mathrm{H}(d)$.

Post Condition verification of authenticity of the received data element, d' such that $\mathrm{H}(d') = \mathrm{H}(d)$.

4.3.4 Delete

An owner of a data element wanting a data element to become unavailable may use the Delete operation. As the blockchain is append-only, it is not possible to remove the previous entry stating the availability of the data element, rather the operation publishes a new entry stating that the data element is unavailable. The storage nodes will delete the data element. The necessary pre- and post-conditions are defined as follows:

Pre Condition there is a transaction committed to the blockchain claiming the availability of some data element d identified by $\mathrm{H}(d)$ for which the client is identified as an *owner*, and all storage nodes store a data element d identified by $\mathrm{H}(d)$.

Post Condition new transaction claiming the unavailability of $\mathrm{H}(d)$ is committed to the blockchain, and that no storage node hold any data element d identified by $\mathrm{H}(d)$.

4.3.5 Verify Storage

Long time archival storage is challenging because over the course of time the data will have to be administered and moved to new hardware as the old is no longer suitable. This means that data loss and data corruption may happen due to management errors or hardware failure, which could go undetected, at least until the data is accessed. This is a major challenge, as large amounts of data may rarely, if ever, be accessed. In addition, accessing entire files is expensive in terms of I/O, especially since the performance of magnetic disks has not improved as rapidly as its capacity [45]. For this reason, the storage must be able to provide non-reputable proofs of their data possession on demand, without accessing the entire file.

Due to using a permissioned blockchain with controlled membership and without having any monetary reward, the storage node can not attempt Sybil-, outsourcing- or generation-attacks [13].

To assess the probability that a client can retrieve its data at some random time in the future, an important consideration in this probabilistic scheme is how many samples are needed for every challenge. We deem a good probabilistic guarantee to have at least 99% chance to detect a modification of 1%. In the following equations we let n denote the total number of file blocks, t the number of faulty blocks, c the number of unique samples in the challenge, X

the discrete random variable which represents the number of samples found to be faulty and finally P_x the probability that at least one faulty block is found. A similar derivation can be found in [17].

$$P_x = 1 - P\{X = 0\} = 1 - \prod_{d=0}^{c-1} \frac{n-t-d}{n-d} \tag{4.1}$$

If we allow for repetition in the samples, we can simplify this to;

$$P_x = 1 - P\{X = 0\} = 1 - \left(\frac{n-t}{n}\right)^c \tag{4.2}$$

$$1 - P_x = \left(\frac{n-t}{n}\right)^c \tag{4.3}$$

$$c = \frac{\log 1 - P_x}{\log\left(\frac{n-t}{n}\right)} \tag{4.4}$$

If we input the numbers for detecting a 1% modification with 99% probability, we end up with a minimum of 459 samples, as shown in Equation 4.5.

$$c = \frac{\log 1 - 0.99}{\log\left(\frac{99}{100}\right)} = 458.21. \tag{4.5}$$

As seen, the minimum number of samples required to achieve a 99% probability to detect a modification of 1% is independent of the total number of file blocks n. Therefore we believe that creating challenges which span several files would be advantageous. In addition, an interesting property of the schemes in [17, 18] is *public verifiability*, which in the context of the schemes means that anyone can issue storage challenges, even those who do not possess the stored data. We believe this would further increase the confidence in the system. The necessary pre- and post-conditions are defined as follows:

Pre Condition there is a transaction committed to the blockchain claiming the availability of some data element d identified by $\mathrm{H}(d)$.

Post Condition new transaction proving the availability of data element d is committed to the blockchain.

Table 4.1: Key distribution.

Readers:	
Identity #1	Enc(K,$PU_{\#id1}$)
Identity #2	Enc(K,$PU_{\#id2}$)
Identity #3	Enc(K,$PU_{\#id3}$)

4.4 Data Encryption

A client concerned with data privacy may want to encrypt their data before uploading. It is left to the client to decide which algorithm that is best suited, as the protocol is not influenced by this. When it comes to sharing the content, and key distribution, the *Change Access* operation allows a decryption key to be specified for every additional reader.

Obviously publishing the decryption key directly on the blockchain is not a good idea. Therefore, the default mode is to encrypt the data using a symmetric cipher such as AES, and then encrypt the encryption key once for each additional reader using their respective public keys, as shown in Table 4.1. The symmetric key should be randomly generated and never be re-used. This allows only those in possession of the private key to decipher the data. A potential weakness with such a scheme is that a reader may share the symmetric key with an unauthorized third-party. However, this is not considered as the reader may also share the plaintext data.

Chapter 5

Implementation

This section will explain the prototype developed during the course of this project. Due to the very nature of a prototype, the details may differ from the idealistic view given in Chapter 4. To save valuable time the choice was made to build the implementation on top of existing open-source architecture. For the ledger nodes Tendermint [30, 31, 33] was used, and for the storage nodes IPFS [28, 32] was used. The first section will give an overview how each of the five operation required to operate the system was implemented. To aid in the explainations, a sequence diagram will be presented for each operation. The next sections will describe inner workings of the prototype in more detail. To augment the textual descriptions, algorithms in pseudocode will be presented. The programming language used in this project is Go. The complete implementation code may be found in Appendix A.

5.1 Operations

Due to the blockchain being append-only, each time we want to change the state we have to append a new transaction to the ledger. We design the transactions in such a way that only the latest committed transaction should be considered when determining the state of a data element. Throughout the detailed description of the operations this will be referred to as retrieving the prevailing block, meaning the block which contains the last transaction that affects the data element.

5.1.1 Upload Data

Figure 5.1 shows the sequence diagram for the *Upload* operation. The protocol starts by having the client upload their data to any storage node. As noted in Section 4.4, the protocol is not influenced by whether the data is encrypted or not. For this reason, data encryption is seen as an orthogonal issue, and is not implemented in this prototype. Further work in Chapter 7 discusses the implementation of data encryption in more detail.

Once the storage node has received the data, it will verify that the sender is a registered client of the system. In addition it will verify that the reported CID and data length is the same as the sender claims. Lastly it will check that the CID is not already stored by the system. The reasoning behind the duplicate check is due to a limitation of the prototype, as it does not support multiple owners for the same data. Given all the conditional verifications pass, the storage node will return a digitally signed envelope containing the users transaction. Though this does not make it impossible to falsely report uploaded data to the ledger node, it forces them to provide a non-reputable proof of the transaction.

After the client receives confirmation from the storage node, it will relay this confirmation to a single ledger node. As discussed in Sections 3.2 and 5.4, the ledger node will verify the transaction and then relay it to the other ledger nodes to achieve consensus for its validity. Both the client and storage node will now wait until the new block containing the upload transaction is committed, at which point the client will generate and save metadata related to the uploaded data, and the storage node will issue the *Pin*-operation in order to distribute the uploaded data to the other storage nodes.

As discussed in Chapter 4, the responsibility of issuing storage challenges is left up to the client. Therefore the next natural step would be to proceed to the verify storage operation, but this is optional.

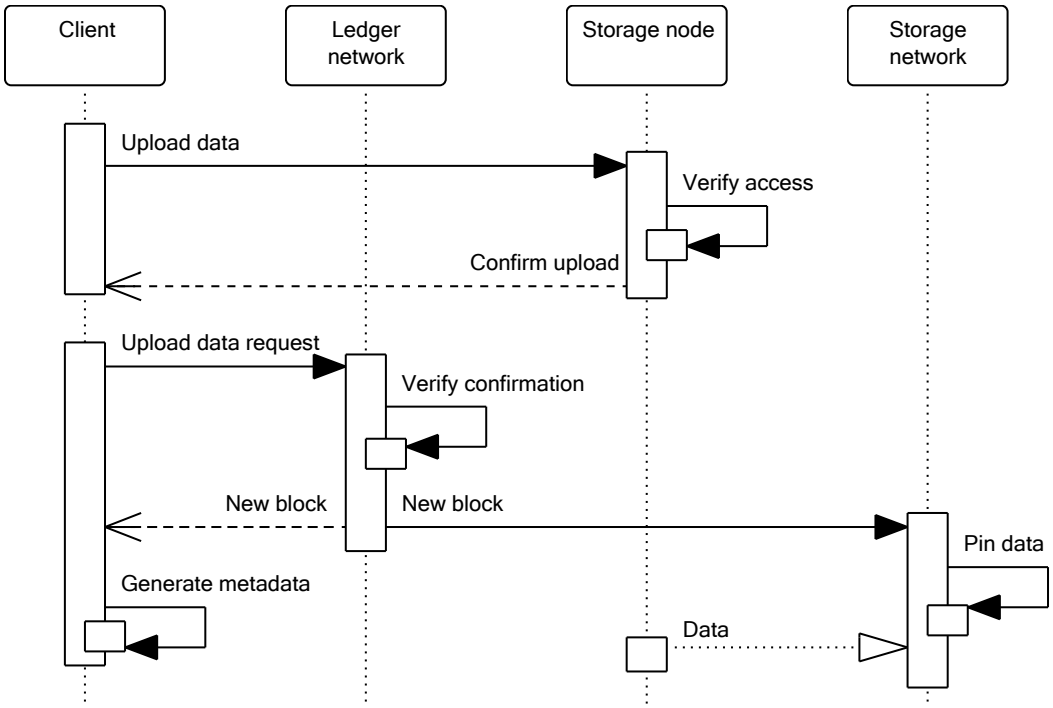

Figure 5.1: Sequence flow for the upload operation.

5.1.2 Download Data

When the client wants to download some piece of stored data, it will issue a signed request containing the data's CID to a storage node. The storage node will verify that the sender of the transaction is a registered client of the system. Then it will fetch the prevailing block for the CID, check the validity of the block, and then verify that the sender has access as a reader of this data. See Algorithm 3 for the details on access verification.

Given that all verifications pass, the storage node will send the data back to the client. The client can then easily verify the integrity of the data by matching its hash to the CID. Figure 5.2 shows a sequence diagram for the download operation. As the figure shows, the download operation does not change the system's state, so there is no need to achieve consensus amongst the nodes. At the same time, it would be easy for a single malicious storage node to withhold data. The protocol imposes no limitation on the number of download requests a client may issue, and to which storage nodes, therefore if a client experiences a non-compliant storage node, it may simply issue the same request to another.

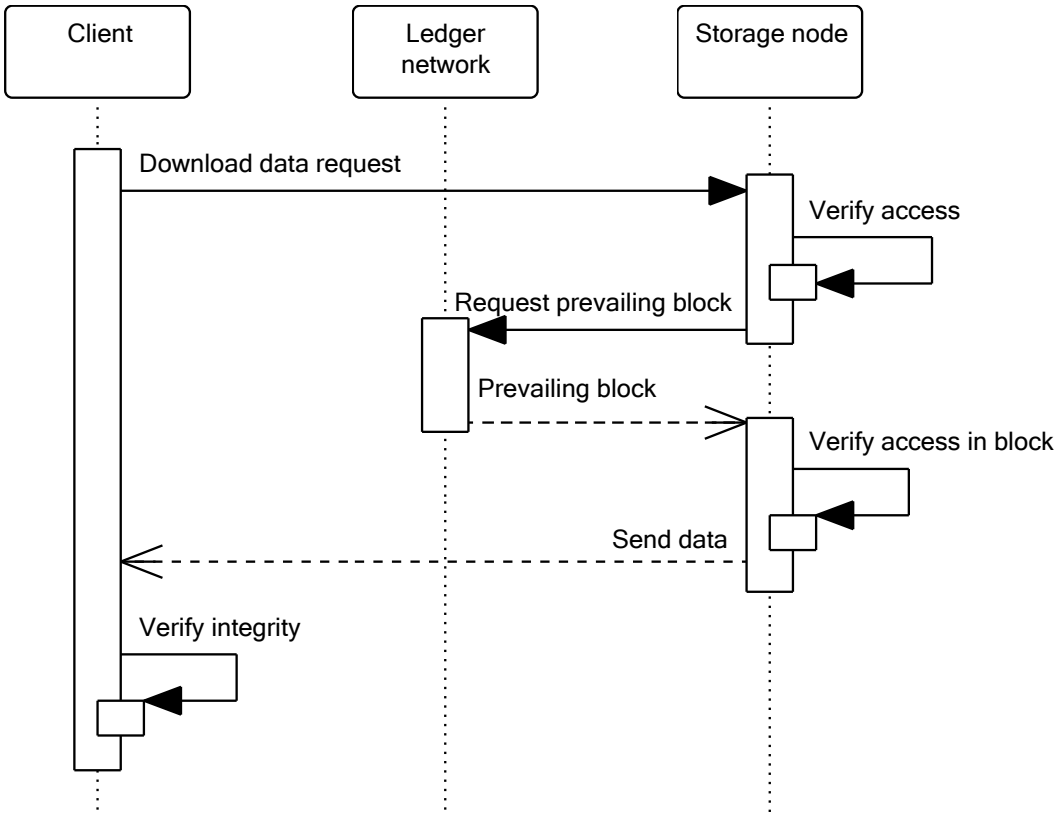

Figure 5.2: Sequence flow for the download operation.

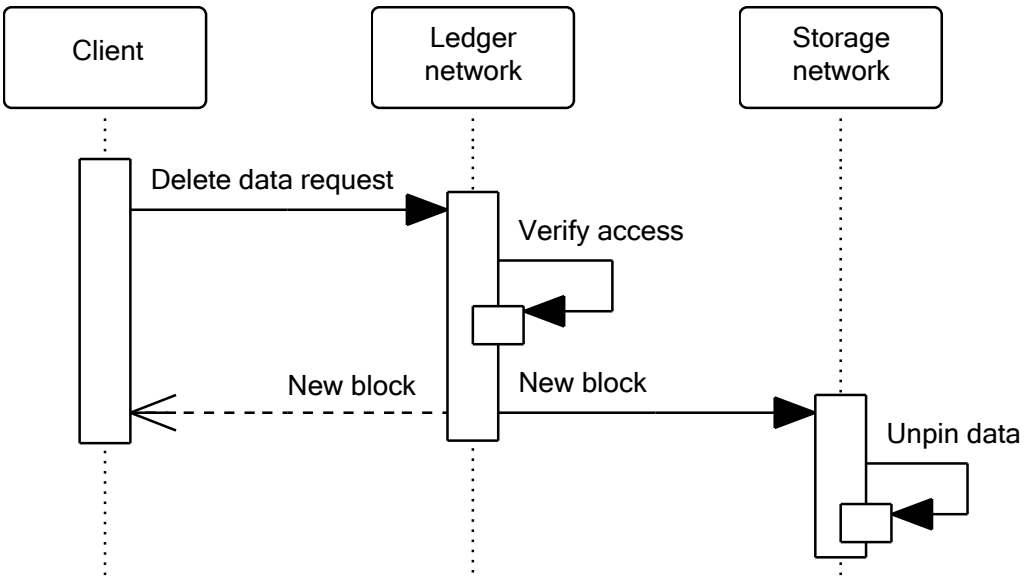

Figure 5.3: Sequence flow for the delete data operation.

5.1.3 Delete Data

The delete data operation starts by having the client issue a signed request containing the CID it wants to delete from the system. The ledger node which receives the request will verify the signature, and that the sender is registered as a client in the system. Then it will use the *isOwner* function as described in Algorithm 3 to verify that the sender is the owner of this data. Given that these condition pass, the ledger node will add the transaction to its mempool and broadcast it to the other ledger nodes. Once a new block containing this transaction is committed to the ledger, all storage nodes will Unpin the data, so that it is no longer guaranteed to stay in persistent storage. Figure 5.3 shows the sequence diagram for the operation.

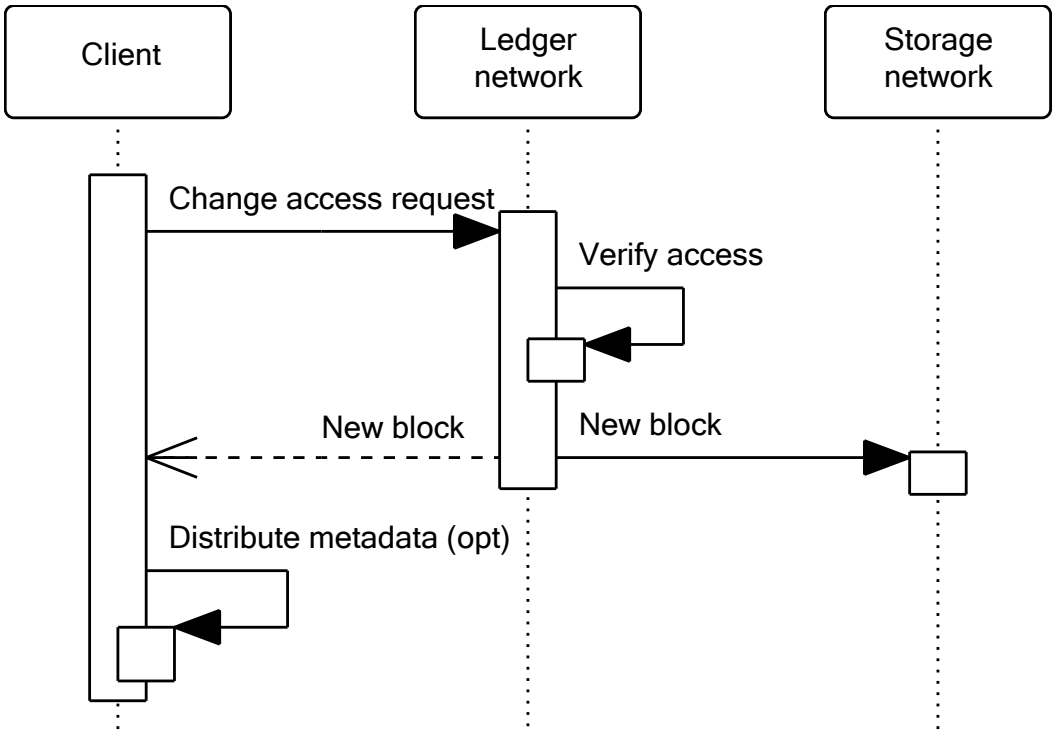

Figure 5.4: Sequence flow for the change content access operation.

5.1.4 Change Access

Figure 5.4 shows a sequence diagram for the change access operation. The steps are identical to the delete data operation up until a new block is committed. The difference being that after a block containing a transaction which changes access on a CID is broadcast, no storage nodes will take any action. The client might however want to distribute the metadata and/or encryption keys for any new readers to the data it might have added. This is however optional, and is left out of this prototype.

5.1.5 Verify Storage

The sequence diagram for the verify storage operation is shown in Figure 5.5. The implementation in the prototype is a simplified version of the one described in [17], with the main difference being that raw content data is stored as part of the metadata. The public verifiability described in Section 2.8.1 is left for future work.

Though not strictly necessary to execute the protocol, normally the client will have uploaded some data and generated and stored its metadata prior to issuing a storage challenge. Algorithm 2 shows the pseudocode for the *sampleData* method, and the rest of the proof-of-storage implementation.

Given that the client has stored the metadata, a challenge is generated using the *generateChallenge* method. The method starts by loading the metadata which contains the samples into a variable. It will then randomly pick out the desired number of samples to fill a challenge, placing the indices in one array, and the values in another array. The array of indices is the generated challenge, and the hash of the array of values will be the proof.

The generated challenge is then signed by the client and sent as a query to a single ledger node. Upon receiving the challenge, the ledger node will initially verify the origin of the challenge, and given that the sender is recognized as a client, it will generate another challenge to be sent to the storage nodes. In order to generate a random challenge, the ledger node needs to know the length of the data. This was previously recorded on the blockchain when the data was initially uploaded. The generation is accomplished using the *generateRandomChallenge* method, which simply returns an array of random integers.

Then both the clients challenge and the ledgers challenge are broadcast to every storage node in the system. The storage node will generate a proof of storage with *proveChallenge*. This method will put the bytes of the stored data in the same order as requested by the challenge and then hash it. The reason for using a cryptographic hash to represent the proof instead of the raw data bytes as suggested in the strawman solution in [17] is that it is designed to be a one-way function, and thus the proof can be put on the blockchain without sacrificing the clients privacy.

The ledger node will collect responses from storage nodes into an array as they come in. Storage nodes have a maximum amount of time in which they will have to respond to the challenge, else they will be considered faulty. Once the ledger node receives the expected number of responses, it will move

on with the protocol. The array of proofs will be signed by the ledger node and broadcast as a transaction. Given that the signatures can be verified by the other ledger nodes, this transaction will be included into the mempool and ultimately contained in a block.

Once the new block containing the proof is committed, the client will verify the proofs that the storage nodes responded with. First it will check if the proofs of the challenge it generated previously is as expected. For the randomly generated challenge of the ledger node it will check if all of the storage nodes responded with the same proof. For the challenge the client generated, it is easy to check which storage nodes were able to prove possession, but for the challenge generated by the ledger node it will have to rely on the majority result to identify faulty storage nodes. It is also reasonable to assume that if a storage node knows that it is unable to respond correctly to a challenge, it may simply opt to not respond at all. Therefore the client should also pay attention to which nodes actually did respond.

Note that since the client interacts with a single ledger node using the *query* connection, it will not interact with the consensus engine. If the request was sent as a regular transaction to the mempool, the protocol would have to be drastically changed; firstly because of the randomness of the extended challenge, and also the fact that more than $\frac{2}{3}$ of the ledger nodes would have to receive the exact same response. A single malicious storage node could simply equivocate by responding differently to every ledger node in an attempt to halt the system. The protocol is however reliant on the honesty of a single ledger node to actually relay the challenges, record the proofs and broadcast them in a transaction. Due to the strength of the cryptographic primitives it may not reliably modify the proofs. One of the design goals of proof-of-storage is that it should be computationally cheap to both issue challenges, and to prove them. There is no limitations on the number of challenges a client can issue, so an unsatisfied client may issue a new challenge to other ledger nodes.

If we keep issuing challenges generated from the same pool of samples, sooner or later a malicious prover would be able to reconstruct the entire stored metadata. Of course the challenges with unknown solutions counteract this, but a client might not be satisfied with this. Therefore, in our prototype we find it necessary that a client can expect a pool of samples to be known by the storage nodes no earlier than a year, if one challenge is issued every day. To calculate how many samples should be generated we use the coupon collectors problem [48] which gives us an expected value for how

many samples must be drawn before you have seen all in the pool. Through trial and error, we were able to find that 16400 samples gives just over a year of daily storage challenges, assuming each challenge contains 460 randomly drawn samples. Equation 5.1 shows the calculations.

$$E[T] = N \sum_{i=1}^{N} \frac{1}{i} = 16400 \sum_{i=1}^{16400} \frac{1}{i} = 168629 \tag{5.1}$$

$$t = 168629/460 = 366,58 \tag{5.2}$$

Algorithm: 2 Proof of Storage

Uses:
TrueRandomGenerator, **instance** *trg*.
Cryptography, **instance** *cry*.

upon event ⟨*cor, Init*⟩ **do**
numSamples := *16400*;
chalLength := *460*;

function *sampleData*(*data*) **returns** Struct **is**
sample := ([]);
sample.indices := ∅;
sample.values := ∅;
max := *length*(*data*);
while |*sample.indices*| < *numSamples* **do**
index := *trg.NewInt*(*0, max*);
sample.indices := *sample.indices* ∪ *index*;
sample.values := *sample.values* ∪ *data*[*index*];
return *sample*;

function *generateRandomChallenge*(*CID*) **returns** Array **is**
chal := ∅;
max := *getDataLen*(*CID*);
while |*chal*| < *chalLength* **do**
chal := *chal* ∪ *trg.NewInt*(*0, max*);
return *chal*;

```
function generateChallenge(CID) returns [Array, String] is
    chal := ∅;
    proof := ∅;
    sample := loadSample(CID);
    while |chal| < chalLength do
        index := trg.NewInt(0, length(sample.values));
        proof := proof ∪ sample.values[index];
        chal := chal ∪ sample.indices[index];
    return [chal, cry.hash(proof)];

function proveChallenge(challenge, data) returns String is
    proof := ∅;
    forall index ∈ challenge do
        proof := proof ∪ data[index];
    return cry.hash(data);
```

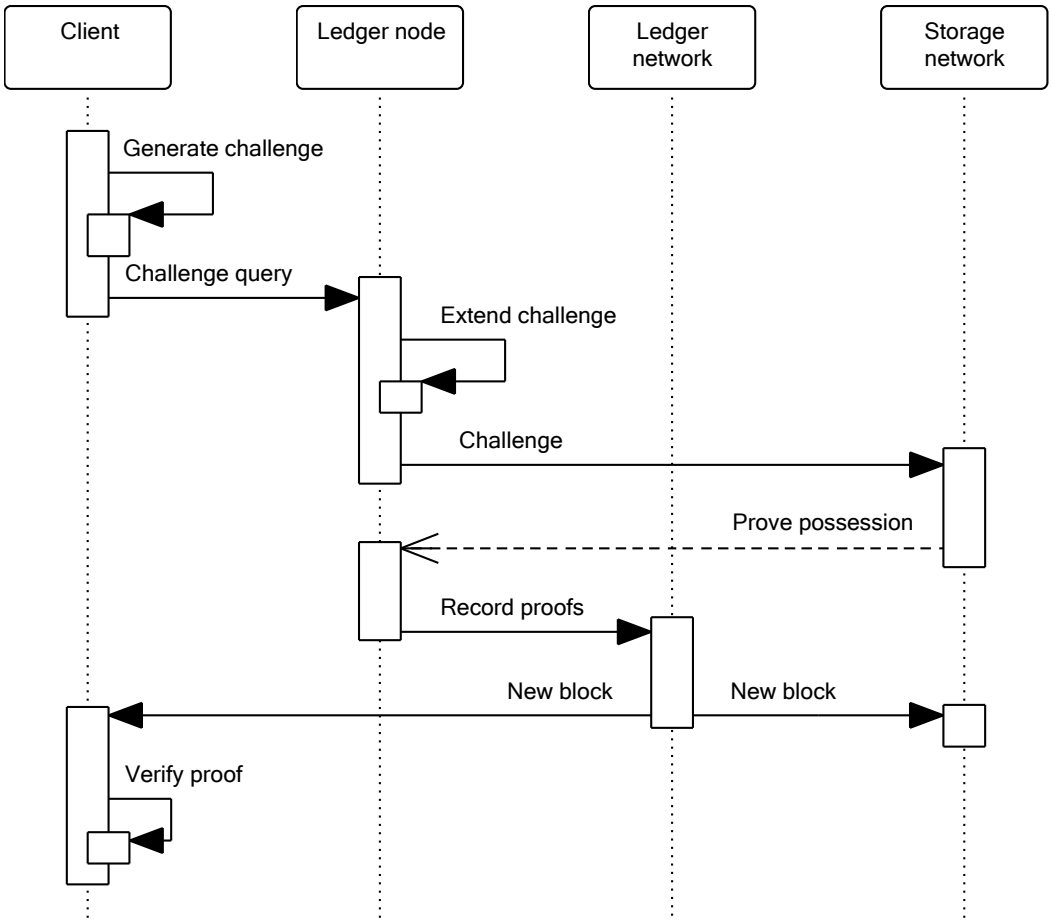

Figure 5.5: Sequence flow for the verify storage operation.

5.2 Access Control

Membership in the prototype is defined by the distribution of public keys, and the configuration file which associates the public key with an access level. There are 3 access levels; *Client*, *Storage node*, and *Ledger node*. Each member is attached to the 128-bit MD5 [19] checksum of the public key. The motivation for using MD5 even though it is no longer considered collision resistant [20] is that it's short and therefore easily readable. In a production environment, a stronger hashing function such as SHA256 should be considered.

The algorithm used for generating the private and public key pair is expected to be RSA [9]. In addition the public key has to be encoded using the DER [40] format, and the private key in the PKCS #8 syntax [41]. The prototype does not enforce any bounds on the key length, though in our tests 4096 bit was used for all keys. At the time of writing this is well above NIST's minimum recommended key length for achieving 128-bit security strength [42] for RSA. In Appendix A an installation guide which details how suitable keys can be generated can be found in the same archive as the program code.

Algorithm 7 gives a simplified pseudocode of how the prototype uses digital signatures. Algorithm 3 gives the pseudocode for how a client's access towards a specific CID is determined. The prototype defines three levels of content access; *None*, *Reader* and *Owner*. The owner is defined to always be the one that originally uploaded the data. A reader is one or several identities that is allowed to download the data, but have no administrative access towards the data. Note that there is not added any functionality to distribute the locally stored metadata between an owner and the readers. Additionally, as data encryption is considered an orthogonal issue, encryption keys must also be shared in some other way.

Algorithm: 3 Access verification

```
function isOwner(block, identity, CID) returns Boolean is
  forall tx ∈ block do
    if CID ≠ tx.CID then
      continue;
    if tx.type = UPLOAD ∧ identity = tx.identity then
      return TRUE;
```

```
      else if tx.type = DELETE then
         return False;
      else if tx.type = ACCESS ∧ identity = tx.identity then
         return True;
   return False;

function isReader(block, identity, CID) returns Boolean is
   forall tx ∈ block do
      if CID ≠ tx.CID then
         continue;
      if tx.type = UPLOAD ∧ identity = tx.identity then
         return True;
      else if tx.type = DELETE then
         return False;
      else if tx.type = ACCESS then
         if identity = tx.identity ∨ identity ∈ tx.readers then
            return True;
   return False;
```

5.3 Storage Node

This section will describe the implementation of the storage node. A storage node consists of three main components; IPFS and IPFS-Cluster developed by Protocol Labs, as well as a proxy built on top to both restrict access to the API and to add additional functionality. IPFS and IPFS-Cluster are discussed in Section 3.1. This section will give a detailed explanation of the proxy component, and how it relates to the other two. A simple illustration is shown in Figure 5.6.

The main motivation for using IPFS in the prototype is that all data is content-addressable (CAS) by its unique, immutable CID. In the context of having an immutable link to the blockchain this is much preferred over location-addressing, as this would limit the flexibility of churn of storage nodes. As noted in Section 3.1, the Unpin operation does not explicitly delete the data, however by adding yet another layer on top of IPFS-Cluster, the data will not be available for download, as long as the node is not acting

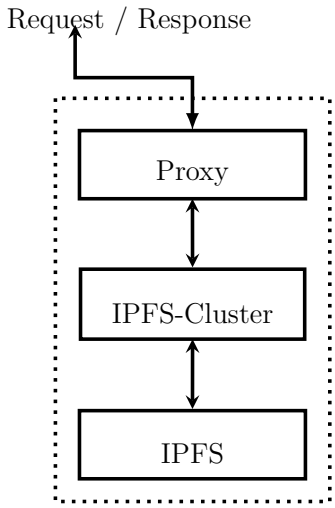

Figure 5.6: Storage node architecture.

maliciously.

Algorithm 4 gives the pseudocode of the proxy component of the storage node. The node receives updates about new blocks being committed to the ledger using the publish-subscribe pattern. The current prototype will only subscribe to a single ledger node, which is quite vulnerable as this ledger node may act malicious by not publishing any new blocks, thereby making the storage node lag behind. An obvious solution to this would be to concurrently subscribe to more than f ledger nodes, or to have the storage node partake as a non-validator.

Once the storage node hears of a new block, it will immediately validate the cryptographic primitives of the block. Note that the signatures contained in a block actually validate the previous block in the chain [31], therefore a more conservative approach would be not to trust any block which does not have at least one valid parent. The storage node will keep track of the last block height it processed so that every block will be processed in order. If it hears of a block which has a height that is more than one higher than the previous processed one, it will request the missing blocks before processing.

The processing of a block is rather simple, as it mainly involves the iteration of the transactions contained in the block. If the transaction is of type *UPLOAD*, the storage will attempt to Pin that CID. Similarly if the type is *DELETE*, the node will attempt to Unpin that CID. After iterating all the

transactions, the last processed height variable is updated.

The proxy offers five endpoints in total; *Get*, *Status*, *StatusAll*, *AddNoPin*, and *Challenge*. The endpoints Status and StatusAll are only accessible for registered clients of the system and are used to get a status report of the stored data. The Get endpoint is used when a client requests to download data. The proxy will only let a client download data if it can verify that it has the role of reader at minimum. When a client wants to upload data, it will send it to the storage node using the AddNoPin endpoint. The storage node will verify that the reported CID and data length is equal to what the client claims. Given this holds, it will store the data in temporary storage until the CID is recorded on the blockchain, at which point it will Pin the data. Finally there is the Challenge endpoint, which is used for generating proofs of storage and used in the Verify Storage operation.

Algorithm: 4 Storage node (IPFS Proxy)

Uses:
IPFSCore, **instance** *ico*.
IPFSCluster, **instance** *icl*.
LedgerCore, **instance** *cor*.
Cryptography, **instance** *cry*.
AccessVerification, **instance** *acv*.
ProofOfStorage, **instance** *pos*.

upon event ⟨*pro, Init*⟩ **do**
identity = *cry.loadCertificate*(*self*);
seenBlockHeight = *0*;
clients, ledgernodes, storagenodes := $[\perp]^N$;
LoadIdentities(*clients, ledgernodes, storagenodes*);

upon event ⟨*cor, NewBlock* | *block* ⟩ **do**
if *validate*(*block*) ≠ TRUE **then**
return;
if *block.height* > seenBlockHeight + 1 **then**
trigger ⟨*cor, RequestBlocks* | [*seenBlockHeight*]⟩
return;
else if *block.height* ≠ seenBlockHeight + 1 **then**
return; // Previously processed.

```
    forall tx ∈ block do
        if tx.type = UPLOAD ∧ identity = tx.storNode then
            icl.Pin(tx.CID);
        else if tx.type = DELETE then
            icl.Unpin(tx.CID);
    seenBlockHeight := block.height;

upon event ⟨pro, Get | node, [tx] ⟩ do
    if hasAccess(tx, clients) ≠ True then
        return;
    block := getPrevailingBlock(tx.CID);
    if validate(block) ∧ acv.isReader(block, tx.identity, tx.CID) then
        trigger ⟨node, GetResponse | [self, tx, ico.getData(tx.CID)]⟩

upon event ⟨pro, Status | node, [tx] ⟩ do
    if hasAccess(tx, clients ∪ ledgerNodes) ≠ True then
        return;
    trigger ⟨node, StatusResponse | [self, icl.status(tx.CID)]⟩

upon event ⟨pro, StatusAll | node, [tx] ⟩ do
    if hasAccess(tx, clients ∪ ledgerNodes) ≠ True then
        return;
    trigger ⟨node, StatusAllResponse | [self, icl.statusAll()]⟩

upon event ⟨pro, AddNoPin | node, [tx, data] ⟩ do
    if hasAccess(tx, clients) ≠ True then
        return;
    if tx.CID ≠ ico.hash(data) ∨ tx.length ≠ data.length then
        return;
    if ico.status(tx.CID) = False then
        ico.addNoPin(data);
        trigger ⟨node, AddNoPinResponse | [self,cry.sign([tx.CID,tx.length],identity)]⟩

upon event ⟨pro, Challenge | node, [tx] ⟩ do
    if hasAccess(tx, clients ∪ ledgerNodes) ≠ True then
        return;
    data := ico.getData(tx.CID);
    proof := pos.proveChallenge(tx.challenge, data);
```

trigger ⟨*node, ChallengeProof* | [SELF,*cry.sign*([*tx.CID,challenge, proof*],*identity*)]⟩

function *hasAccess*(*tx, openFor*) **returns** Boolean **is**
return *cry.verifysignature*(*tx*) ∧ *tx.identity* ∈ *openFor*;

function *getPrevailingBlock*(*cid*) **returns** Block **is**
trigger ⟨*cor, Query* | [SELF, *'/prevailingheight', tx.CID*] ⟩;
upon event ⟨*self, QueryResponse* | *cor,* [*height*] ⟩ **do**
trigger ⟨*cor, GetBlock* | [SELF, *height*] ⟩;
upon event ⟨*self, BlockResponse* | *cor,* [*block*] ⟩ **do**
return *block*;

5.4 Ledger Node

This section describe how the Tendermint application was implemented. In particular, we will focus on the consensus methods *CheckTx* and *DeliverTx*, as well as the query methods *Challenge* and *Prevailingheight*. Algorithm 5 gives the pseudocode for the implementation.

To establish communication with the consensus engine (Tendermint Core), Tendermint supports two solutions [31]. The first, and default one is raw sockets and the second is gRPC. The choice for the prototype fell on gRPC as this made is easier to both create new and extend existing message types. The voting power for each validator was kept uniform.

To prevent denial of service attacks on the mempool [33], we implement filtering on which transactions are valid. In addition, by adding unique sequence numbers to transactions, we can employ *exactly-once* semantics to prevent replay attacks.

When implementing the CheckTx and DeliverTx methods, it is crucial that their execution is deterministic [31]. Depending on the severity of the non-determinism it may be impossible for the protocol to reach consensus. The safest approach is to avoid all sources of non-determinism. This includes, but is not limited to: true random generators, any form of race conditions and reliance on the system clock. It is also important to consider the details

of the programming language, as some sources of non-determinism may be more subtle, such as map iteration in Go. This is the reasoning behind the *sort* operation in the *stringify* method of Algorithm 7, as it sorts all the keys in lexicographic order.

Any time a new transaction is validated against CheckTx, it will first verify the digital signature of the transaction, and as discussed in Section 5.2, that the sender is a member of the system. Given the verification passes, the rest of the validation depends on the type of transaction. The *Verify* type is the most simple, as the only additional condition is that the sender must have the ledger node access level. The *Delete* and *Access* types both require that the sender is the owner of the content. The *Upload* type requires that the sender has the client access level. If the type-dependent check passes, CheckTx will return an *OK* response, which means that the transaction is valid, and can be included in the mempool.

As elaborated on in Chapter 7, the *BeginBlock* and *EndBlock* have been omitted in this prototype, and therefore we are only concerned with *DeliverTx* once a new block is committed. For every new block, the transactions will be iterated and DeliverTx executed one time for each. The executed code depends on the type of transaction. For the *Delete* and *Access* types, the map which keeps track of block height is relevant for any CID will be updated. Transactions of the *Verify* type have no additional logic. When it comes to transactions with type *Upload*, simple metadata containing the file size of the uploaded data will be written to persistent storage. The file size is later used to define the interval of the random challenges. Note that by having the client collude with the storage node it is possible to trick the ledger nodes into committing a block containing erroneous CID or file size, as the uploaded data never passes through the ledger node. However, this would immediately be detected after the first proof-of-storage challenge, as a valid proof holds both the CID and file size. Due to the non-repudiation property of digital signatures, it would be trivial to prove which client and storage nodes private key signed the transaction.

Tendermint offers a *Query* connection, which allows the interaction with the associated application without engaging consensus. Queries are filtered by their *path*. The prototype contains two query paths; *prevailingHeight* and *challenge*. *Prevailingheight* is by far the simplest one, as it will simply return the block height which should be considered when processing a given CID. Note that, as stated in Section 5.3, simply querying a single ledger node is prone to malicious behavior. An alternative solution would be to have all

actors keep up with the blockchain and thus keep this data structure themselves. The *challenge* path is used when a client wants to issue a proof of storage request for some previously uploaded data. The input to the challenge query must be a digitally signed storage challenge by a registered client. In addition to the clients storage challenge, the ledger node will generate a fully random challenge - of which it does not know the proof for. The reasoning behind the random challenge is discussed in Section 5.1.5. The two challenges are then issued to all the storage nodes known to the ledger node. The execution will then halt until enough storage proofs have been returned, or the requests have timed out. All the proofs are then collected, and then digitally signed in a transaction and broadcast as a new transaction to the network. Once this transaction is recorded on the ledger, everyone can see what proofs the storage nodes responded with. Note that due to the assumed strength of the cryptographic primitives, the single ledger node may not modify the proofs. The ledger node may however act malicious by simply not broadcasting the proofs, therefore the client should issue the same challenge to multiple ledger nodes. In the current implementation it is also possible for the client to issue storage challenges directly to the storage node, but these would not be recorded on the ledger.

Algorithm: 5 Ledger node (Tendermint App)

Uses:
LedgerCore, **instance** *cor*.
ProofOfStorage, **instance** *pos*.
AccessVerification, **instance** *acv*.
Cryptography, **instance** *cry*.

upon event ⟨*app, Init*⟩ **do**
prevailingBlock := $[\perp]^N$;
clients, ledgernodes, storagenodes := $[\perp]^N$;
LoadIdentities(*clients, ledgernodes, storagenodes*);
proofs := $([]^N)$;

upon event ⟨*app, ChallengeProof* | *stornode,* [*proof, chal*] ⟩ **do**
if *cry.verifysignature*(*proof*) ≠ TRUE ∨ stornode ∈ *proofs*[*chal*] **then**
return;
append(*proofs*[*chal*], [*stornode, proof*]);

```
upon event ⟨app, ChallengeProof | stornode, [TIMEOUT, chal] ⟩ do
   append(proofs[chal], [stornode, TIMEOUT]);

function Query(path, DATA) returns String is
   if path = '/challenge' then
      if cry.verifysignature(DATA.tx) ≠ TRUE ∨ DATA.tx.identity ∉ clients
            then
         return '';
      rndChal := pos.generateRandomChallenge();
      proofs := ([]^N);
      forall storNode ∈ storageNodes do
         trigger ⟨storNode, Challenge | [DATA.tx.challenge]⟩;
         trigger ⟨storNode, Challenge | [rndChal]⟩;
      upon |proofs[DATA.tx.challenge]| = |storageNodes| ∧ |proofs[rndChal]|
            = |storageNodes| do
         trigger ⟨Broadcast | [NEWTX, proofs]⟩;
         return 'Proofs recorded';
   else if path = '/prevailingheight' then
      return prevailingBlock[DATA.CID];
   else
      return '';

function queryStor(node, DATA, endpoint) returns String is
   trigger ⟨node, Proxy | [ENDPOINT, DATA]⟩;
   upon event ⟨app, StorageResponse | node, [response] ⟩ do
      return response;

function CheckTx(tx) returns Boolean is
   return cry.verifysignature(tx) ∧ validateTx(tx);

function DeliverTx(tx) returns Boolean is
   return doTxWork(tx);

function validateTx(tx) returns Boolean is
   if tx.type = UPLOAD then
      if tx.identity ∉ clients ∨
            queryStor(tx.storNode, tx.CID, status) ≠ TRUE then
```

```
        return FALSE;
      return TRUE;
   else if tx.type = DELETE then
      block := prevailingBlock[tx.CID];
      return acv.isOwner(block, tx.identity, tx.CID);
   else if tx.type = ACCESS then
      block := prevailingBlock[tx.CID];
      return acv.isOwner(block, tx.identity, tx.CID);
   else if tx.type = VERIFY then
      return tx.identity ∈ ledgerNodes;
   else
      return FALSE;

function doTxWork(tx) returns Boolean is
   if tx.type = UPLOAD then
      pos.WriteSimpleMetadata(tx.CID, tx.length);
      prevailingBlock[tx.CID] = cor.height + 1;
      return TRUE;
   else if tx.type = DELETE then
      prevailingBlock[tx.CID] = cor.height + 1;
      return TRUE;
   else if tx.type = ACCESS then
      prevailingBlock[tx.CID] = cor.height + 1;
      return TRUE;
   else if tx.type = VERIFY then
      return TRUE;
   else
      return FALSE;
```

5.5 Client

The pseudocode for a client who can interact with the system is shown in Algorithm 6. In addition to the the five operations discussed in Section 5.1, it has three more commands; *View blockchain*, *List metadata* and *Storage*

node status. As they don't involve consensus, these are omitted from the pseudocode in Algorithm 6, but can be found in the attached program code in Appendix A.

It is not essential that a client connects to the current proposer, as once a transaction is passed into the mempool, it will be gossiped to through the mempool reactor to be cached in the mempool of all the validators [33]. Any validator which has the transaction in its mempool may propose it in due time. One could however argue that preferential broadcasting to the next proposer may reduce the block commit time.

When the client wants to upload data, it will input its local path into the application, including optional fields for name and description. The name and description fields are only used as part of the locally stored metadata. The CID and length of the data is then calculated and digitally signed, and then sent alongside the data to a storage node. Once the client receives a response from the storage node that the upload was successful, it will verify the validity of the response and then relay it to a ledger node. Note that only the storage nodes response is sent to the ledger node, and not the actual file data. The client will subscribe to the publication of new blocks, and once the upload transaction is committed, it will generate a storage sample of the data, and store it alongside the metadata.

To issue a storage challenge, the client will have to input the CID of the data it wants to challenge. Then *generateChallenge* of Algorithm 2 will use the previously stored sample to generate a storage challenge. The challenge is signed and then sent to a ledger node in a query. When a new block which contains all the storage nodes proofs for this challenge is committed, the client will verify that the proofs are correct. Should the proof be incorrect, or some storage node did not respond, it would be appropriate for the client to either issue a new challenge or trigger some kind of *suspect* protocol. This is however left to further work.

The other operations mainly involve signing the application input and then relaying it either a storage node or a ledger node.

Algorithm: 6 Client

Implements:
BCFSClient, **instance** *cli*.

Uses:

IPFSProxy, **instance** *ipo*.
LedgerCore, **instance** *cor*.
Cryptography, **instance** *cry*.
ProofOfStorage, **instance** *pos*.
IPFSCore, **instance** *ico*.

upon event ⟨*pro, Init*⟩ **do**
identity = *cry.loadCertificate*(*self*);
challenges := $([]^N)$;
uploads := $([]^N)$;

upon event ⟨*cor, NewBlock* | *block* ⟩ **do**
forall *tx* ∈ *block* **do**
if *tx.type* = *VERIFY* ∧ *tx.challenge* ∈ *challenges* **then**
myProof := *challenges*[*tx.challenge*];
forall *proof* ∈ *tx.proof* **such that** *proof* ≠ *myProof* **do**
suspect(*proof.identity*);
else if *tx.type* = *UPLOAD* ∧ *tx.CID* ∈ *uploads* **then**
sample := *pos.sampleData*(*uploads*[*tx.CID*]);
pos.saveMetadata(*tx.CID, sample*);

upon event ⟨*cli, Challenge* | *CID* ⟩
[*chal, proof*] := pos.generateChallenge(cid);
append(*challenges*[*chal*], *proof*);
trigger ⟨*cor, Query* | [SELF, *'/challenge',cry.sign*([*CID, chal*], *identity*)] ⟩;

upon event ⟨*cli, Upload* | *data* ⟩
CID := ico.hash(data);
tx := cry.sign([CID, length(data)], identity);
trigger ⟨*ipo, AddNoPin* | [SELF, *tx, data, identity*] ⟩;
upon event ⟨*ipo, AddNoPinResponse* | *ipoTx* ⟩ **do**
if *cry.verifysignature*(*ipoTx*) ∧ *ipoTx.CID* = *CID* ∧ *ipoTx.length* = *length*(*data*) **then**
append(uploads[tx.CID], data);
trigger ⟨*cor, UPLOAD* | [NEWTX, *ipoTx*]⟩;

upon event ⟨*cli, Get* | *CID* ⟩
tx := *cry.sign*([*CID*], *identity*);

```
    trigger ⟨ipo, Get | [SELF, tx] ⟩;

upon event ⟨ipo, GetResponse | q, [NODE, tx, data] ⟩
    if tx.CID := ico.hash(data) then
        saveData(tx.CID, data);

upon event ⟨cli, Delete | CID ⟩
    tx := cry.sign([CID], identity);
    trigger ⟨cor , DELETE | [NEWTX, tx] ⟩;

upon event ⟨cli, Access | [CID, readers] ⟩
    tx := cry.sign([CID, readers], identity);
    trigger ⟨cor, ACCESS | [SELF, tx] ⟩;
```

5.6 Datastructures

Figure 5.7 shows some of the data structures used in the prototype. At the top we have *SignedStruct*, which is used to represent digitally signed data. For better or worse, it uses an empty interface in the *Base* field to hold the value of any type. Note that *Signature* holds the signature for the hash of Base, not the value of Base itself. Two of the types that the prototype use in Base are *StorageChallenge* and *Transaction*.

Transaction is the data structure used to represent all requests to the ledger nodes, and may be contained in a block on the blockchain. It also has an empty interface that can be used to hold the value of any type. In the most simple cases it will simply hold a string, whilst in other cases it holds complex structures such as an array of storage proofs. Some data structures also have another field, *Nonce*, which is used to deter replay attacks, and provides exactly-once semantics. Note however, that the implementation of replay-mitigation in the prototype does not store anything to persistent storage, and is therefore vulnerable after a restart of the node.

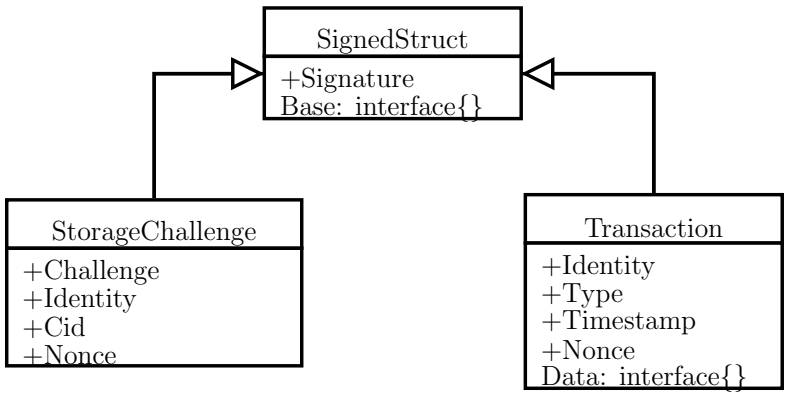

Figure 5.7: Datastructures used in the prototype.

5.7 Cryptography

This section contains pseudocode for the cryptographic primitives used in the prototype.

Algorithm: 7 Cryptography

Uses:
RivestShamirAdleman, **instance** *rsa*.
SecureHashAlgorithm, **instance** *sha256*.

function *verifysignature*(*tx*) **returns** Boolean **is**
return *rsa.verify*(*hash*(*tx.base*), *tx.signature*, *getPubKey*(*tx.identity*));

function *sign*(*data*, *identity*) **returns** String **is**
return *rsa.sign*(*hash*(*data*), *getPrivKey*(*identity*));

function *stringify*(*data*) **returns** String **is**
if |*fields*(*data*)| = *0* **then**
return data;
output := ";
forall *field* ∈ *sort*(*data*) **do**
output := *output* + *stringify*(*field*);
return *output*;

function *hash*(*data*) **returns** String **is**
return *sha256*(*stringify*(*data*));

Chapter 6

Evaluation

We attempt to answer research question **RQ2** by evaluating scalability of the prototype using 3 different configurations of quorum sizes. First we have a small network with 7 ledger nodes and 3 storage nodes, then a medium network with 16 ledger nodes and 9 storage nodes and finally a large network with 25 ledger nodes and 15 storage nodes. To evaluate, we test how long it takes for the network to disseminate data after the client uploads it. Three different data sizes are used, 10MB, 100MB and 500MB. We do 20 experiments for each data size in every configuration. To issue requests to the network we use a single client. The complete experimental data can be found in Appendix B.

6.1 Experimental Setup

Experiments are conducted on virtual machines on Amazon Elastic Compute Cloud. The virtual machines reside in the same data center in the Frankfurt region. We spread the virtual machines out in the three availability zones offered to reduce the likelihood that multiple virtual machines will reside on the same physical server. The virtual machines are configured as *t2.medium*, which has 2 vCPU, 4 GiB of RAM and 30 GiB of SSD storage. The operating system used is *Ubuntu Server 16.04 LTS (HVM)*. Tests are run using the following versions; Tendermint 0.19.2, IPFS 0.4.15-rc1, IPFS-Cluster 0.3.5 and Go 1.10.1.

To configure the network, we generate a public/private key pair for every node. The public key is shared with every other node on the network. To

configure the ledger nodes, we list all of them as validators and give them uniform voting power upon genesis of the blockchain. The storage nodes are configured to have the same secret key to ensure secure communication [32].

In order to reduce errors in the measurements we dedicate one server as a Network Time Protocol (NTP) server [68], and then make all the nodes synchronize their local time to the server. The accuracy of the nodes can be kept within one millisecond in a LAN environment [69]. As our measurements of Pin time are only recorded in seconds, the time difference between nodes is several orders of magnitude lower and should thus not impact the results.

6.2 Data Dissemination Results

For each measurement we start the time as soon as the client initiates the Upload operation. The first data point is block commit time, then every contiguous data point is when a storage node has completed the Pin operation. Figures 6.1, 6.2, 6.3 show the mean time for data dissemination in a small, medium and large network, respectively. As seen in Table 6.1, the time before all storage nodes have completed the Pin operation increase far more rapidly than the time before a quorum of nodes have completed the operation. This is due to the last few nodes often taking a long time before completing the operation. This is also shown in the error bars of the figures. Indeed, in one instance the Pin operation ran for over 1 hour before printing an error message. As this is most likely due to a bug, we have excluded this result from the measurements. We also noticed a time discrepancy between a storage node having the data element in storage and completing the Pin operation. However, we were unable to perform any measurement to confirm this finding.

As expected, the number of ledger nodes did not impact data dissemination speed. As noted in Chapter 5, a block may be committed as soon as one storage node reports having stored the data. This means that the data transfer speed between the client and the storage node is the dominant factor.

The number of ledger nodes would likely affect performance if multiple Upload operations where concurrently being executed. Experimental results for throughput and latency for Tendermint can be found in [33], which states that Tendermint can achieve thousands of transactions per second with up to 64 ledger nodes.

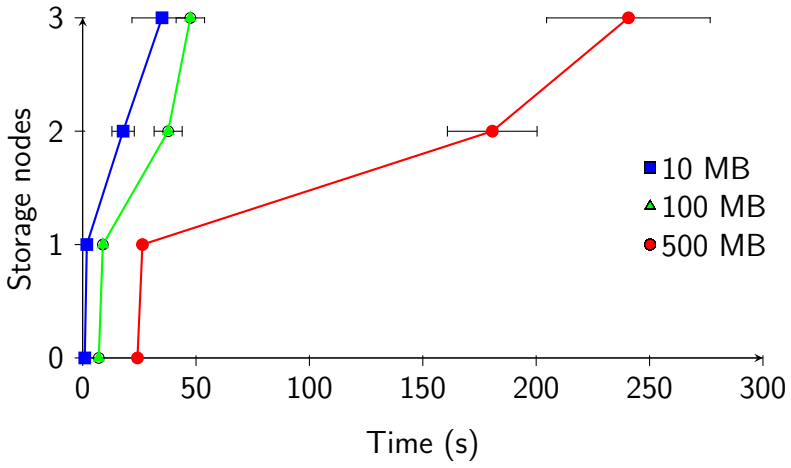

Figure 6.1: Data dissemination after initiating an upload in a small network consisting of 7 ledger nodes and 3 storage nodes. Each data point is the mean of 20 experiments. The horizontal error bars show the standard error.

Data size 10 MB		
Network size	Quorum Pin	Full Pin
Small	17.84	34.99
Medium	29.92	88.42
Large	28.63	141.88
Data size 100 MB		
Network size	Quorum Pin	Full Pin
Small	37.71	47.50
Medium	61.42	102.12
Large	98.66	169.86
Data size 500 MB		
Network size	Quorum Pin	Full Pin
Small	180.67	240.72
Medium	287.65	378.50
Large	277.23	521.33

Table 6.1: Mean time in seconds for data dissemination for different network sizes. Quorum Pin means once over half of the storage nodes has the data in persistent storage.

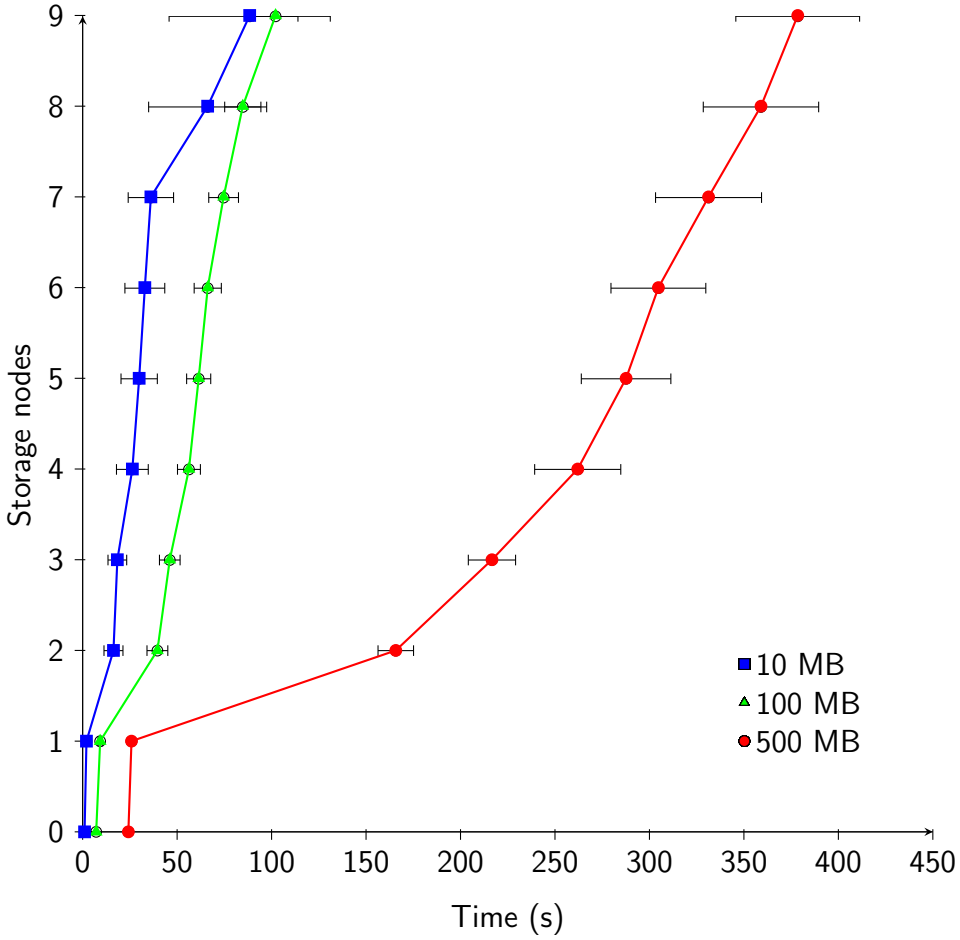

Figure 6.2: Data dissemination after initiating an upload in a medium network consisting of 16 ledger nodes and 9 storage nodes. Each data point is the mean of 20 experiments. The horizontal error bars show the standard error.

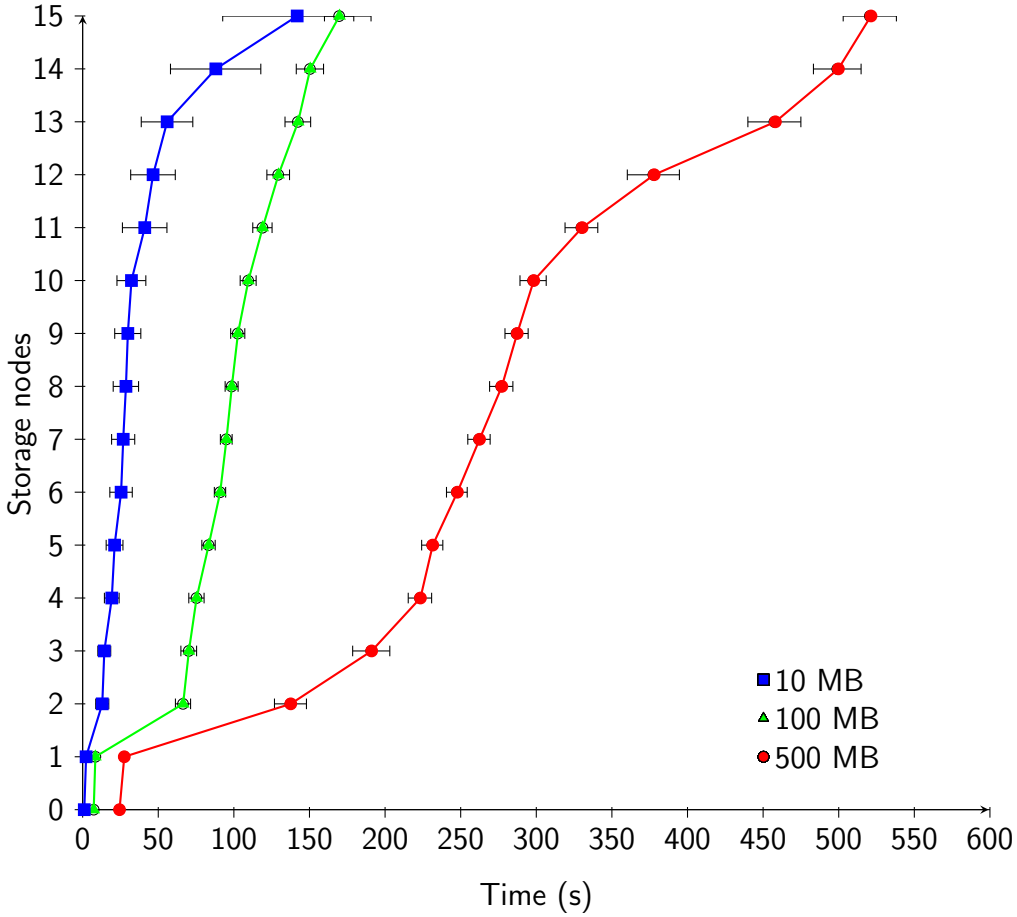

Figure 6.3: Data dissemination after initiating an upload in a large network consisting of 25 ledger nodes and 15 storage nodes. Each data point is the mean of 20 experiments. The horizontal error bars show the standard error.

Chapter 7

Further work

This chapter provides some suggestions and thoughts for improvement for the work presented in this thesis.

Audit Log By modifying the Download operation to publish an entry into the ledger, the system would be able to provide chronological documentation of each data access. Such a feature might be of particular interest for a system dealing with medical records or financial transactions.

Data Encryption The client component should offer support for encrypting data using some symmetric cipher. To allow for multiple readers of the encrypted data, the encryption key should be encrypted using the readers public key, and then published on the blockchain. Additionally, means for a system operator to hold a master key should be studied further.

Secure Signing Devices To hold the clients public/private key pair and digitally sign data, a secure signing device could be used. Additionally, this could require some biometric input such as a fingerprint to be activated.

Non-Validators The storage nodes should subscribe to multiple ledger nodes in order to reliably receive updates of new blocks being committed. As an alternative in Tendermint, the storage node could be added as a non-validator.

Availability Numbers The availability numbers for each component should be studied further, for example by modeling them in a Markov model.

Public Verifiability Of Storage Nodes The prototype uses a primitive proof-of-storage algorithm which only allows owners of the stored data to issue storage challenges. Using a more sophisticated approach as described in Section 4.3.5 would allow anyone to issue storage challenges.

Simultaneous Storage Challenges Currently each storage challenge is done on a per-CID level. If a challenge could span multiple CIDs, this would reduce the amount of data being published on the blockchain.

Smart Contracts By adding the possibility of clients specifying smart contacts as part of the operations, advanced features and great flexibility could be gained. The appropriate execution model for smart contracts should be studied further. Some of the features that could be specified are time limited sharing of data and requiring multiple parties to sign an operation (e.g. delete).

Suspect Protocol Currently, it is possible to detect erroneous storage proofs by looking at the blockchain. These detections should be used to suspect and notify a system operator of a potential faulty storage node. Additionally, it should be possible to trigger a suspect protocol if either a storage node is refusing to accept uploaded data, or sends erroneous data when downloading.

Multiple Owners Currently the prototype expects exactly one owner for each CID. Further study is required to devise a way to support multiple owners uploading the same data.

Dynamic Membership In the prototype each participant is given their membership view upon initialization. Compared to ledger- and storage nodes, we expect clients to have by far the highest churn rate, and therefore want to add functionality which allows for modification in client membership without requiring the restart of nodes. In addition, we think a study of repair mechanisms after ledger- or storage node failure would be interesting.

Chapter 8

Conclusion

This project initially had a very large scope with several difficult tasks, so this thesis should only be seen as preliminary work for the project as a whole. In this thesis we were able to reduce the scope of the project, and focus on the main task which was to design a highly scalable architecture for a storage system built on blockchain technology that requires strong data integrity. In addition, we identified three specific research questions, **RQ1**, **RQ2** and **RQ3** which we addressed in this thesis.

During the course of this project a deep study of literature surrounding blockchains was undertaken. On the basis of this study we were able to identify and classify the desired characteristics of a storage system built on blockchain technology. A design for a scalable architecture was presented, and we found five basic operations needed to operate the system. By organizing the architecture into three distinct components, we were able to separate the agreement protocol from the storage servers [43, 44]. This in turn allowed for reduced redundancy of the storage servers, which is highly desirable in an economic sense.

We were able to answer research questions **RQ1** and **RQ3** by finding that by using a permissioned blockchain and controlling membership the need for resistance against Sybil attacks is alleviated. This allows for the usage of more traditional BFT consensus algorithms which can offer higher update and query rates, and instantaneous transaction confirmation, without sacrificing integrity. The drawback is that the number of writers does not scale well as communication overhead becomes crippling [53, 12]. In addition, permissioned blockchains needs to have a way of reconfiguring membership.

A prototype based on the system architecture was implemented. The

prototype is only the first implementation of the architecture, and there is still considerable work to be done. This includes architectural changes, added functionality, formal proofs and performance optimizations. By running experiments on the prototype, we were able to empirically confirm the number of ledger and storage nodes supported, and thus answer research question **RQ2**. Further work remains to more accurately determine upper limits for ledger and storage nodes, and indeed the upper limit for clients in the peer-to-peer network.

The prototype was built upon the existing open-source blockchain architecture, Tendermint [30], and the distributed file system, IPFS [28]. Both of these projects are under continuous development, and especially in the case of Tendermint, the development cycle may include breaking changes [31]. Due to this, using the latest version proved a futile effort, and specific release branches were used instead.

On a positive note, during the course of the project my learning outcome was great as I was able to learn a great deal about the state of the art of blockchain systems and several related technologies.

Appendix A

Program Code

The full program code for the prototype is attached to this thesis. In the attachment an installation guide can be found in *README.md*. There is also a version controlled repository that was used in the development process.

Attached program code: 
Repository: `https://github.com/racin/DATMAS_2018_Implementation`

Appendix B

Complete Experimental Data

This appendix contains the complete experimental data obtained from the experiments conducted in Chapter 6. For each of the 9 experiment configurations, 20 measurements were taken.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	0.94	0.10	0.80	1.10	0.90	0.02
1	1.84	0.35	0.80	2.10	1.90	0.08
2	17.84	22.03	1.80	63.90	4.95	4.93
3	34.99	59.19	1.80	246.90	11.60	13.24

Table B.1: Results for data dissemination of a 10 MB file in a small network consisting of 7 ledger nodes and 3 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	7.10	0.14	6.90	7.40	7.10	0.03
1	8.91	3.84	7.10	25.20	8.10	0.86
2	37.71	27.70	21.10	130.10	29.15	6.19
3	47.50	28.01	26.90	130.10	37.10	6.26

Table B.2: Results for data dissemination of a 100 MB file in a small network consisting of 7 ledger nodes and 3 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	24.17	0.28	23.80	24.70	24.15	0.06
1	26.37	2.35	24.80	32.60	25.40	0.53
2	180.67	88.45	109.30	403.70	153.15	19.78
3	240.72	161.40	129.20	801.80	178.65	36.09

Table B.3: Results for data dissemination of a 500 MB file in a small network consisting of 7 ledger nodes and 3 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	1.02	0.15	0.80	1.30	1.00	0.03
1	2.02	0.15	1.80	2.30	2.00	0.03
2	16.32	22.46	1.90	75.80	5.35	5.02
3	18.37	22.13	2.90	76.80	8.45	4.95
4	26.32	37.61	2.90	163.20	12.95	8.41
5	29.92	42.96	2.90	164.20	13.95	9.60
6	32.92	47.20	2.90	182.20	16.45	10.55
7	36.12	53.69	2.90	189.20	16.45	12.01
8	66.17	139.72	2.90	497.20	21.00	31.24
9	88.42	190.80	2.90	653.20	23.50	42.66

Table B.4: Results for data dissemination of a 10 MB file in a medium network consisting of 16 ledger nodes and 9 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	7.17	0.24	6.80	7.50	7.25	0.05
1	9.27	4.91	7.80	30.10	8.30	1.10
2	39.57	24.64	21.30	116.10	29.70	5.51
3	46.12	24.42	22.30	116.10	36.80	5.46
4	56.27	27.11	31.40	118.10	42.30	6.06
5	61.42	28.51	33.80	121.10	50.00	6.38
6	66.22	32.15	33.80	130.20	51.40	7.19
7	74.62	35.21	33.80	140.20	61.70	7.87
8	84.77	42.84	34.80	163.80	64.70	9.58
9	102.12	52.97	47.30	205.10	79.60	11.85

Table B.5: Results for data dissemination of a 100 MB file in a medium network consisting of 16 ledger nodes and 9 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	24.20	0.29	23.80	24.70	24.15	0.06
1	25.85	1.75	24.80	32.30	25.35	0.39
2	165.75	42.04	108.10	240.80	145.80	9.40
3	216.65	55.93	137.30	331.20	198.90	12.51
4	262.05	101.93	165.30	489.10	214.90	22.79
5	287.65	105.99	180.30	533.10	265.95	23.70
6	304.75	112.27	190.30	560.10	266.45	25.10
7	331.30	125.45	200.10	598.10	284.70	28.05
8	359.05	136.59	207.10	644.10	295.50	30.54
9	378.50	146.51	222.10	664.10	310.00	32.76

Table B.6: Results for data dissemination of a 500 MB file in a medium network consisting of 16 ledger nodes and 9 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	1.08	0.18	0.80	1.40	1.05	0.04
1	2.28	0.51	1.80	3.30	2.15	0.11
2	12.98	19.37	2.20	80.20	5.05	4.33
3	14.33	19.09	2.90	80.20	7.45	4.27
4	19.28	21.80	3.90	83.20	10.60	4.87
5	21.13	24.86	3.90	102.20	13.35	5.56
6	25.43	33.21	3.90	131.20	14.35	7.43
7	26.83	34.24	3.90	134.20	15.35	7.66
8	28.63	37.55	3.90	142.20	15.60	8.40
9	29.88	38.64	3.90	142.20	16.35	8.64
10	32.28	42.76	3.90	157.20	17.45	9.56
11	41.08	65.92	3.90	252.20	18.95	14.74
12	46.58	66.07	4.90	255.20	20.85	14.77
13	55.88	76.34	7.90	321.20	29.00	17.07
14	88.08	133.75	9.90	588.20	38.90	29.91
15	141.88	219.68	10.90	951.20	51.55	49.12

Table B.7: Results for data dissemination of a 10 MB file in a large network consisting of 25 ledger nodes and 15 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	7.31	0.26	6.90	7.90	7.20	0.06
1	8.41	0.48	7.90	9.50	8.20	0.11
2	66.46	22.47	25.60	105.40	64.60	5.02
3	70.26	23.28	28.60	107.20	62.80	5.20
4	75.36	22.63	32.60	108.20	69.85	5.06
5	83.36	19.79	48.60	111.40	83.70	4.42
6	90.96	16.71	54.90	117.20	89.25	3.74
7	95.11	17.39	55.90	121.20	97.30	3.89
8	98.66	19.17	56.90	137.20	99.30	4.29
9	102.71	20.95	56.90	141.20	102.05	4.68
10	109.61	23.59	57.90	151.20	110.05	5.28
11	119.06	28.56	57.90	166.00	117.40	6.39
12	129.51	33.56	62.90	199.00	130.10	7.50
13	142.51	37.99	76.90	242.00	141.65	8.49
14	150.51	40.39	81.90	261.00	147.15	9.03
15	169.86	43.38	90.90	280.00	166.65	9.70

Table B.8: Results for data dissemination of a 100 MB file in a large network consisting of 25 ledger nodes and 15 storage nodes. Unit in seconds.

Times for data dissemination (s)						
Storage nodes	Mean	SD	Min	Max	Median	SE
Block commit	24.38	0.29	23.90	24.90	24.40	0.06
1	27.53	3.39	25.00	33.80	25.70	0.76
2	137.63	47.36	32.50	217.90	142.70	10.59
3	191.13	55.00	82.50	301.90	207.85	12.30
4	223.38	34.64	153.50	306.90	223.45	7.75
5	231.53	31.37	175.50	308.90	229.80	7.01
6	247.83	30.68	193.50	323.20	244.20	6.86
7	262.48	33.08	197.50	329.90	255.40	7.40
8	277.23	34.47	218.50	341.20	276.90	7.71
9	287.38	34.32	232.50	357.20	290.15	7.67
10	298.33	38.83	236.50	363.20	298.95	8.68
11	330.33	48.47	252.40	401.40	314.00	10.84
12	377.98	77.16	260.40	521.10	389.85	17.25
13	458.13	78.50	313.50	591.40	448.70	17.55
14	499.78	70.56	388.00	621.40	515.00	15.78
15	521.33	78.82	390.00	667.50	525.65	17.62

Table B.9: Results for data dissemination of a 500 MB file in a large network consisting of 25 ledger nodes and 15 storage nodes. Unit in seconds.

Bibliography

[1] A. Narayanan, J. Bonneau, E. Felten, A. Miller, S. Goldfeder, J. Clark, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*, Draft - Feb 9, Princeton, 2016. [Online]. Available: `https://lopp.net/pdf/princeton_bitcoin_book.pdf`.

[2] A. Narayanan, J. Bonneau, E. Felten, A. Miller, *Bitcoin and Cryptocurrency Technologies*, 2015. Available: `https://www.coursera.org/learn/cryptocurrency`. Accessed on: 01/10/2018.

[3] A. M. Antonopoulos, *Mastering Bitcoin: Programming the Open Blockchain*, 2nd ed, Sebastopol: O'Reilly Media, Inc., 2017.

[4] C. Cachin and M. Vukolić, "Blockchain Consensus Protocols in the Wild", *CoRR*, arXiv:1707.01873v2 [cs.DC], 2017.

[5] A. Narayanan, J. Clark, "Bitcoin's Academic Pedigree", *acmqueue*, vol. 15, no. 4, Aug. 2017, doi: 10.1145/3132259, [Online].

[6] M. Hearn, *Corda: A distributed ledger*, Version 0.5, 2016. [Online]. Available: `https://docs.corda.net/_static/corda-technical-whitepaper.pdf`.

[7] R. G. Brown, J. Carlyle, I. Grigg, M. Hearn, *Corda: An Introduction*, 2016. [Online]. Available: `http://blockchainlab.com/pdf/corda-introductory-whitepaper-final.pdf`.

[8] M. Vukolić, "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication", presented at iNetSec 2015: Open Problems in Network Security, 2016.

[9] W. Stallings, *Cryptography And Network Security: Principles and Practice*, 6th ed., Pearson Education, 2014.

[10] C. Cachin, R. Guerraoui, L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer, 2011.

[11] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382-401, Jul. 1982, [Online]. Available: `https://people.eecs.berkeley.edu/~luca/cs174/byzantine.pdf`.

[12] B. Arati, "Understanding Blockchain Consensus Models", Persistent Systems Ltd, Apr. 2017, [Online]. Available: `https://www.persistent.com/wp-content/uploads/2017/04/WP-Understanding-Blockchain-Consensus-Models.pdf`.

[13] Protocol Labs, "Filecoin: A Decentralized Storage Network", Aug. 2017, [Online]. Available: `https://filecoin.io/filecoin.pdf`.

[14] Protocol Labs, "Proof of Replication - Technical Report (WIP)", Jul. 2017, [Online]. Available: `https://filecoin.io/proof-of-replication.pdf`.

[15] K. Wüst, A. Gervais, "Do you need a Blockchain", 2017, IACR Cryptology ePrint Archive (2017): 375, [Online]. Available: `https://eprint.iacr.org/2017/375.pdf`.

[16] H. Meling, K. Marzullo, A. Mei, *When you don't trust clients: Byzantine proposer fast paxos*, In: Proceeding of ICDCS, pp. 193-202, Jun. 2012, doi: 10.1109/ICDCS.2012.38.

[17] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, "Provable data possession at untrusted stores", In: De Capitani di Vimercati, and P. Syverson, editors, *ACM CCS 07*, pp. 598-609, ACM Press, Oct. 2007.

[18] H. Shacham, B. Waters. "Compact Proofs of Retrievability", In *Advances in Cryptology - ASIACRYPT '08*, Springer, pp. 90-107, 2008.

[19] R. Rivest, *The MD5 Message-Digest Algorithm*, RFC 1321, Apr. 1992. [Online]. Available: `https://tools.ietf.org/pdf/rfc1321.pdf`, Accessed on: 03/04/2018.

[20] Microsoft, *Flame maleware collision attack explained*, Security Research & Defense, Microsoft TechNet Blog, Jun. 2012. [Online]. Available: `https://blogs.technet.microsoft.com/srd/2012/06/06/flame-malware-collision-attack-explained/`, Accessed on: 03/04/2018.

[21] C. Boutin, *NIST Releases SHA-3 Cryptographic Hash Standard*, NIST, Aug. 2015. [Online]. Available: `https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard`. Accessed on: 03/04/2018.

[22] L. Lamport, "Paxos Made Simple", ACM SIGACT News (Distributed Computing Column), 32(4):18-25, Dec. 2001. [Online]. Available: `https://lamport.azurewebsites.net/pubs/paxos-simple.pdf`.

[23] J.-P. Martin, L. Alvisi, "Fast Byzantine Consensus", IEEE TODSC, 3(3):202-215, Jul. 2006. [Online]. Available: `http://www.cs.cornell.edu/lorenzo/papers/Martin06Fast.pdf`.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications", In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[25] H. D. Johansen, R. V. Renesse, Y. Vigfusson, D. Johansen, "Fireflies: A secure and scalable membership and gossip service". ACM Transactions on Computer Systems (TOCS), 33(2):5:1-5:32, May 2015.

[26] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, R. Wattenhofer, "On Scaling Decentralized Blockchains", pp. 106-125, Springer Berlin Heidelberg, Berlin, Heidelberg, Aug. 2016, doi: 10.1007/978-3-662-53357-4_8.

[27] G. Badishi, I. Keidar, A. Sasson, "Exposing and Eliminating Vulnerabilities to Denial of Service Attacks in Secure Gossip-Based Multicast". In *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, pp. 201-210, Jun.-Jul. 2004.

[28] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)", arXiv:1407.3561, Jul. 2014. [Online]. Available:

`https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf`.

[29] R. Rodrigues, P. Druschel, "Peer-to-Peer Systems", *Communications of the ACM*, vol. 50, no. 10, pp. 72-82, Oct. 2010, doi:10.1145/1831407.1831427.

[30] J. Kwon, "Tendermint: Consensus without Mining", Aug. 2014, [Online]. Available: `https://tendermint.com/static/docs/tendermint.pdf`.

[31] Tendermint Documentation. [Online]. Available: `http://tendermint.readthedocs.io/projects/tools/en/master/`, Accessed on: 03/11/2018.

[32] A guide to running ipfs-cluster, Version: 0.3.1. [Online]. Available: `https://github.com/ipfs/ipfs-cluster/blob/master/docs/ipfs-cluster-guide.md`, Accessed on: 03/21/2018.

[33] E. Buchman, "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains". Master's thesis, University of Guelph, Jun. 2016.

[34] M. Castro, B. Liskov, "Practical Byzantine Fault Tolerance". In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, New Orleans, Feb. 1999.

[35] R. Pass, E. Shi, "Thunderella: Blockchains with Optimistic Instant Confirmation". In *Proceedings of Eurocrypt*, 2018.

[36] C. Dwork, N. Lynch, L. Stockmeyer, "Consensus in the presence of partial synchrony", In *Journal of the ACM*, vol. 35, no. 2, pp. 288-323, Apr. 1988.

[37] Hyperledger Fabric Documentation. [Online]. Available: `https://media.readthedocs.org/pdf/hyperledger-fabric/latest/hyperledger-fabric.pdf`, Accessed on: 05/31/2018.

[38] D. Ongaro, J. Ousterhout, "In search of an understandable consensus algorithm", In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*, pp. 305-319, Jun. 2014.

[39] R. Nygaard, DATMAS_2018_Implementation (source code). `https://github.com/racin/DATMAS_2018_Implementation`.

[40] International Telecommunication Union, "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", Jul. 2002, [Online]. Available: `https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf`, Accessed on: 05/12/2018.

[41] B. Kaliski, *Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification Version 1.2*, RFC 5208, May. 2008. [Online]. Available: `https://tools.ietf.org/html/rfc5208`, Accessed on: 05/12/2018.

[42] E. Barker, *NIST Special Publication 800-57 Part 1, Revision 4*, Jan. 2016. [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf`, Accessed on: 05/12/2018.

[43] C. Cachin, D. Dobre, M. Vukolić, *Separating Data and Control: Asynchronous BFT Storage with 2t + 1 Data Replicas*. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 – October 1, 2014. Proceedings*, pp. 1-17, 2014.

[44] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, M. Dahlin. "Separating agreement from execution for Byzantine fault-tolerant services". In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 253-268, Oct. 2003.

[45] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, R. Stutsman'. "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM". In *SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92-105, Dec. 2009.

[46] E. A. Brewer. "Towards robust distributed systems". In *Proc. 19th ACM on Principles of Distributed Computing*, Jul. 2000.

[47] B. Cohen. "Incentives build robustness in BitTorrent". In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, vol. 6, pp. 68-72, May. 2003.

[48] M. Ferrante, M. Saltalamacchia. "The Coupon Collectors Problem", In *MATerials MATemtics*, vol. 2014, no. 2, pp. 1-35, 2014. [Online]. Available: `http://mat.uab.cat/matmat/PDFv2014/v2014n02.pdf`. Accessed on: 05/21/2018.

[49] S, Dziembowski, S. Faust, V. Kolmogorov, K. Pietrzak, "Proofs of space", In: *Gennaro R., Robshaw M. (eds) Advances in Cryptology – CRYPTO 2015, Lecture Notes in Computer Science*, vol 9216, pp. 585-605, Aug. 2015, Springer, Berlin.

[50] Hyperledger Architecture, Volume 1. The Linux Foundation. [Online]. Available: `https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf`. Accessed on: 05/23/2018.

[51] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, C. Montgomery. "Sawtooth: An Introduction". The Linux Foundation, Jan, 2018. [Online]. Available: `https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf`. Accessed on: 05/23/2018.

[52] V. Buterin, V. Griffith. "Casper the Friendly Finality Gadget". arXiv:1710.09437v2, Nov. 2017.

[53] I. Eyal, A. E. Gencer, E. G. Sirer, R. V. Renesse, "Bitcoin-NG: A Scalable Blockchain Protocol". In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI '16*, Mar. 2016.

[54] Y. Sompolinsky, A. Zohar, "Secure High-Rate Transaction Processing in Bitcoin". In *Financial Cryptography and Data Security - 19th International Conference, FC 2015*, pp. 507-527, Jan. 2015.

[55] R. C. Merkle, "A digital signature based on a conventional encryption function". In *Pomerance C. (eds) Advances in Cryptology - CRYPTO'87, Lecture Notes in Computer Science*, vol. 293, pp. 369-378, Springer, Berlin, Heidelberg, 1987.

[56] L. Lamport, "The Part-Time Parliament", In *ACM Transactions on Computer Systems*, 16(2):133-169, May. 1998.

[57] Y. Amir, B. Coan, J. Kirsch, J. Lane, "Byzantine Replication Under Attack", In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, Jun. 2008.

[58] M. J. Fischer, N. Lynch, M. S. Paterson, "Impossibility of distributed consensus with one faulty process", In *Journal of the ACM*, 32(2):374-382, Apr. 1985.

[59] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, E. Cecchet, "ZZ and the Art of Practical BFT Execution", In *Proceedings of the sixth conference on Computer systems*, Eurosys '11, pp. 123-138, Apr. 2011, doi: 10.1145/1966445.1966457.

[60] A. Miller, Y. Xia, K. Croman, E. Shi, D. Song, "The honey badger of BFT protocols", In *Cryptology ePrint Archive 2016/199*, 2016.

[61] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", 2008. [Online]. Available: `https://bitcoin.org/bitcoin.pdf`.

[62] J. R. Douceur, "The Sybil Attack", In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Mar. 2002. [Online]. Available: `https://www.microsoft.com/en-us/research/wp-content/uploads/2002/01/IPTPS2002.pdf`.

[63] J. Sousa, A. Bessani, "From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation", In *Proc. 9th European Dependable Computing Conference*, pp. 37-48, 2012.

[64] A. Bessani, J. Sousa, E. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt", In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014*, pp. 355-362, 2014.

[65] Kafka 1.0 Documentation. Apache. [Online]. Available: `https://kafka.apache.org/10/documentation.html`, Accessed on: 05/31/2018.

[66] D. Vorick, L. Champine, "Sia: Simple Decentralized Storage", Nebulous Inc., Tech. Rep. Nov. 2014. [Online]. Available: `https://coss.io/documents/white-papers/siacoin.pdf`, Accessed on: 06/01/2018.

[67] Nebulous Inc., Sia Wiki, 2018. Available `https://siawiki.tech/index`. Accessed on: 06/01/2018.

[68] D. L. Mills, "Internet Time Synchronization: The Network Time Protocol", In *IEEE Transactions on Communcations*, vol. 39, pp. 1482-1493, Oct. 1991.

[69] U. Windl, et. al. *The NTP FAQ and HOWTO - How does it work?*. [Online] Available: `http://www.ntp.org/ntpfaq/NTP-s-algo.htm`. Accessed on: 06/07/2018.

[70] P. Maymounkov, D. Mazières, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", In *Proceedings of IPTPS02*, Cambridge, USA, Mar. 2002.

[71] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, P. Gerbes, P. Hutchins, C. Pollard, V. Buterin, "Storj A Peer-to-Peer Cloud Storage Network", v2.0, Des. 2016. [Online]. Available: `https://storj.io/storj.pdf`.

[72] M. Conoscenti, A. Vetrò, J. C. D. Martin, "Blockchain for the Internet of Things: a Systematic Literature Review", In *Proceedings of the IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, pp. 1-6, Nov. 2016.

[73] K. Okupski, "Bitcoin Developer Reference - Working Paper", Jul. 2016. [Online]. Available: `http://enetium.com/resources/Bitcoin.pdf`.

[74] I. Eyal, E. G. Sirer, "Majority is not Enough: Bitcoin Mining is Vulnerable", In *Financial Cryptography and Data Security*, 2014.

[75] S. Haber, W. S. Stornetta, "How to Time-Stamp a Digital Document", In *Advances in Cryptology - Crypto '90, Lecture Notes in Computer Science*, vol. 537, pp. 437-455, 1991.

[76] M. Bishop, H. M. Conboy, H. Phan, B. I. Simidchieva, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, S. Peisert, "Insider Threat Identification

by Process Analysis", In *Proceedings of the third IEEE Workshop on Research in Insider Threats, WRIT'14*, pp. 251-264, 2014.