Consensus-less Security

A truly scalable distributed ledger

J. Brouwer, 2020

TU Delft

Delft University of Technology

Consensus-less Security

A truly scalable distributed ledger

by

J. Brouwer

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Friday January 10, 2020 at 10:00.

Student number:	4615964	
Project duration:	November 12, 2018 – January 10, 2020	
Thesis committee:	Dr. ir. J. A. Pouwelse,	TU Delft, supervisor
	Dr. S. Roos,	TU Delft
	Dr. M. Zuniga,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Distributed ledger technology was expected to spark a technical revolution similar to the internet revolution. After the release of Bitcoin in 2008, many developments have significantly increased the performance of distributed ledger technology. Nevertheless, the first truly scalable ledger has yet to be deployed. All of them have issues with scaling in either the throughput, the number of nodes which can validate transaction or both.

The concept behind a distributed ledger is that the integrity of the ledger is a shared responsibility. However, as soon as new technology emerges, also misuse surfaces, especially if there are financial gains involved. The general solution, to prevent such abuse, in distributed ledger technology is through the use of global consensus. If the majority of a network is honest, and we require a majority vote on the validity of a transaction, no malicious transactions will succeed. A downside of requiring a majority vote is that every node eligible to vote must contain full knowledge on all previous transactions. This work argues that the requirement of global consensus is a major limiting factor when it comes to the scalability of current ledgers.

The goal of this work is to design a scalable distributed ledger whose security does not rely on global consensus.

It proposes a novel algorithm that guarantees security, even under adversarial attack, by up to $\frac{1}{3}$ of the network exhibiting byzantine behavior. It does so using Trustchain, a pair-wise ledger designed by the Delft University of Technology, and 'Fair Witness Selection Protocol', a newly designed publicly verifiable witness selection algorithm with an indicated message and communication complexity of $O(log^{\star}(n))$. A mathematical lower-bound is given on the security level of the algorithm, and the security is reduced to the security of the underlying hash function.

Several experiments were executed on the DAS-5 supercomputer to confirm the scalability of this work. These experiments show that the throughput of the network scales linearly, and has been tested up to 2500 nodes (simultaneously acting as validators and clients). To the best of the author's knowledge, it is the only ledger that has no theoretical limits on the number of clients, number of validators, or throughput. A peak-throughput of 7025 tx/s has been observed at a network size of 280 nodes. Furthermore, the total transaction time remained roughly constant at about 15 milliseconds regardless of the network size.

Preface

Welcome, before you lies the embodiment of little over a year's worth of work to obtain the degree of Master of Science. I would like to thank everybody who supported me with everything. Knowing me, I'd probably forget someone, so I'm not going to put names here. But if you feel I owe you one, This one is for you: Thank you. And if not, you're still reading this, meaning you got further than most other people in this world, so thank you too!

I hope something in this thesis will make you go *'Ah, that's clever!'*, or maybe *'what the hell is going on'*, preferably the former, but the latter is fine with me too. (At least I invoked some emotion?). And now as per long standing preface-tradition, to pretend to be more intellectual than I actually am, I shall finish this preface with a quote — *"So long, and thanks for all the fish."*

J. Brouwer
Delft, January 2020

Contents

1

Introduction

Distributed ledger technology has received much attention over the last few years. It promised to be a game-changing technology that will shape the future of the world economy, improving everything from healthcare to online voting [1]. So far, no real revolution has happened. Is this simply the infamous 'Trough of Disillusionment' every new technology goes through after its inception, or are there severe technological issues preventing this promised revolution?

While the Bitcoin paper, released in 2008, sparked a wide interest in digital currencies and distributed ledger technology, it was hardly the first [5]. Already in 1983, Chaum proposed an untraceable digital payment system using blind signatures [6]. A client would create its own banknotes and have them signed by a bank. Since the bills were signed blindly, a bank would not be able to trace the bill, but a receiver would be sure that it is valid due to the bank signature. In the 1990s and early 2000, several companies (such a DigiCash and Peppercoin [3][4]) were founded to bring electronic cash out of the cryptographer's realm and into the real world. However, none of them succeeded.

One of the key issues with digital cash is double-spending: A physical €10 bill can only be owned by a single person at a time, and it can only change ownership but can not be replicated. With digital cash, however, there is no guarantee that after transferring the money, the previous owner deletes the digital €10. This is where Bitcoin's innovation lies. Where some solutions rely on calling the issuing bank to verify that the coin is still valid [7] or having each bank host a public database with all currently valid coins[22], Bitcoin publishes all transactions on a public ledger. However, instead of a single bank being responsible for this ledger, it is a shared responsibility of all users.

The ledger proposed by Nakamoto, commonly referred to as *blockchain*, is essentially a linked list. Transactions are bundled in blocks, and each block is completed by a pointer to the previous block. However, instead of using the block's address as a pointer, it uses the hash of the block as its pointer. By doing so, it becomes a tamper-resistant chain of blocks. Changing the contents of a block will change the hash, and thus breaking any pointers towards the block. Every block is then completed by adding a solution to a cryptographic puzzle. The solver of this cryptographic puzzle is rewarded with Bitcoins, incentivizing people hosting and maintaining the ledger. The process of solving this puzzle is called *mining*.

The difficulty of the cryptographic puzzle is chosen in such a way to keep the time it takes to mine a block artificially high. The probability of multiple parties creating a block at the same time is kept low. If two blocks are created at the same time, a node arbitrary picks one block and considers it valid. After a while, one of those blocks will have a successor. By a simple rule stating that the longest chain is the valid one, eventual consensus will be reached on

which blocks and, therefore, which transactions are valid.

This process of reaching an agreement on which transactions are valid and which are invalid is called *global consensus*. This specific method of reaching consensus is dubbed Nakamoto-consensus after the inventor, but many other consensus algorithms exist. Global consensus aims to address the double-spending problem by reaching consensus on the validity and total order of all transactions; If all participants know all transactions, one cannot hide earlier spending of the funds. While it seems that global consensus does perform adequately when it comes to prevention of double-spending, it does create new issues.

A characteristic property of a distributed system is scalability: Ideally, a distributed system should be extensible without bottlenecks, and the increase of the capacity of a system should be proportional to the extension. However, when requiring global consensus, the number of nodes that need to agree grows linearly with the number of nodes in the network. Unless a hypothetical algorithm with a run-time complexity of $O(1)$ is used, the capacity per node of such a system will only decrease with the addition of more nodes.

Another side-effect of global consensus is that by imposing a total order on all transactions, all nodes should possess information about all transactions. At the time of writing, there are 10412 known Bitcoin nodes, each storing the complete ledger of 201 GB, a total of 2 Petabytes of data stored [8].

But even when assuming an instantaneous consensus algorithm, one that reaches consensus without any added latency, problems persist. Most distributed ledgers technologies employ a single ledger, resulting in multiple parties working on a data structure and all the problems associated with that. All in all, the use of global consensus as a tool to create a secure distributed ledger might not be worth the problems that come with it.

In this thesis, we explore the possibility of creating a secure, distributed ledger while avoiding the use of global consensus. This exploration starts in chapter 2 with a problem description and analysis of the current state of the art. In chapter 3, this work's solution is described, of which a proof of concept is designed and implemented in chapter 4. This proof of concept is then evaluated using experiments, which can be found in chapter 5. All of this is finalized by a conclusion and future work in chapter 6.

2

Background and Problem description

Early on in their education, computer scientists are made aware of all the difficulties global variables bring. Concurrent read and write access to a single data structure can lead to race conditions, requiring mutexes, semaphores, and two-phase locks to prevent them. Yet here we are, the R3 consortium (receiver of the largest single investment for a distributed ledger technology company at $107 million[23]) claims in [24] that it is time to move away from a situation where *"each financial institution maintains its own ledgers, which record that firm's view of its agreements..."* to *"one where a single global logical ledger is authoritative for all agreements between firms recorded on it"*.

From a business perspective, it makes perfect sense to demand a perfectly ordered list of valid transactions on which everyone agrees; If party A and B are in dispute, one can take a look at this globally agreed-upon ledger with all agreements and settle the dispute. However, from a computer science perspective, the requirement of a global data structure introduces many difficulties, of which some are even proven impossible to solve. To understand why these problems exist, it is important to first elaborate on distributed systems and consensus algorithms.

2.1. Distributed Systems

A distributed system is a set of autonomous computers which present themselves as a coherent system which cooperate as a means to an end. There is a plurality of motivations to deploy distributed systems, including but not limited to performance, availability, reliability, extendibility, and even the notion that the humans served by these systems are of a distributed nature themselves.

One of the largest and best known distributed systems, the World Wide Web, displays the performance gain from being distributed. Having a single computer with all the data in the world seems a bad idea, not only regarding availability and security. But attempting to serve the 122.000 petabytes per month of IP traffic as estimated for the internet in 2017[25] from a single machine seems futile. By distributing the load across multiple computers, this performance can be achieved.

The reliability and availability arise from the notion that well-designed distributed systems can keep functioning even when one or more computers fail.

As a result of these favorable properties of distributed systems, distributed ledgers have received increasing interest from enterprises and financial institutions. Since digital infras-

tructure is becoming a more critical aspect of businesses, downtime and performance bottlenecks can have a significant impact on these businesses. However, just as with large groups of people working together, disagreement can occur (be it due to malformed messages, lost messages, delayed information, or malicious behavior), and requires solving before proceeding.

Reaching consensus is a fundamental problem in computer science as it emerges in an extensive range of applications of distributed systems. Applications include database replication, leader election, clock synchronization, but also in a more rudimentary form in flight-computers as they are often redundantly fitted, and only one can be used at a single time.

If nodes would never crash or misbehave, consensus would be trivial. However, real-world processors can crash, and nodes may turn rogue. To aid the design of fault-tolerant distributed systems, failure models are used to put a bound on the number and the type of failures that are allowed to take place before the system stops behaving in a well-defined manner. The type of failures can be classified as permanent or transient faults. Transient faults occur temporary, for example, corruption of memory due to voltage glitches or a failed transmission of messages.

Permanent faults can be further classified into two individual models:

1. *crash failure model*: A processor simply stops at an arbitrary point and will not resume ever again. It is assumed that the failing processor will not corrupt messages; A message is either sent in its entirety or not at all.

2. *Byzantine failure model*: In this model everything may happen. A processor is allowed to arbitrary stop sending or receiving messages or arbitrarily decide to send messages with any content to other processors in the system. This model also includes malicious behavior with the intent to break or abuse the distributed system. In a byzantine failure model 'failing' processors are also allowed to collude with other processors.

The crash failure is an often used failure model for designing algorithms where all the nodes within the system are trusted or honest, but crashes may occur due to network, power, or hardware failures. Byzantine failure models are often used when malicious nodes are to be considered. Since Byzantine fault-tolerant algorithms also include tolerance against knowingly and willingly malicious behavior, they give a higher assurance regarding security but are also more complex. In some cases, Byzantine fault tolerance can even be impossible. An excellent example of this is the three generals problem first publish in [26] by Lamport, Shostak, and Pease.

Imagine three generals, Alice, Bob, and Charles, of which one is a traitor. Each has their army surrounding a city, for them to be successful, they must either attack together or not attack at all. Say that Charles proposes a plan and informs Alice and Bob about the plan, however, Charles is malicious and tells Alice to attack but Bob to retreat. When Alice then confirms with Bob what to do, she gets conflicting information. This scenario is shown in Figure 2.1 on the left.

Now imagine the same situation, but instead of Charles, Bob is the malicious actor. Shown in Figure 2.1 on the right. When Alice asks Bob for confirmation, he lies and says that Charles told to retreat. From Alice's perspective, it is impossible to distinguish between the two situations; therefore, she can not make the correct decision. While the original paper gives a more rigorous proof and generalizes the impossibility to any network of size $3m + 1$ (where m is the number of malicious actors), this example gives an intuition why, in general, less than m parties are considered malicious when designing distributed algorithms.

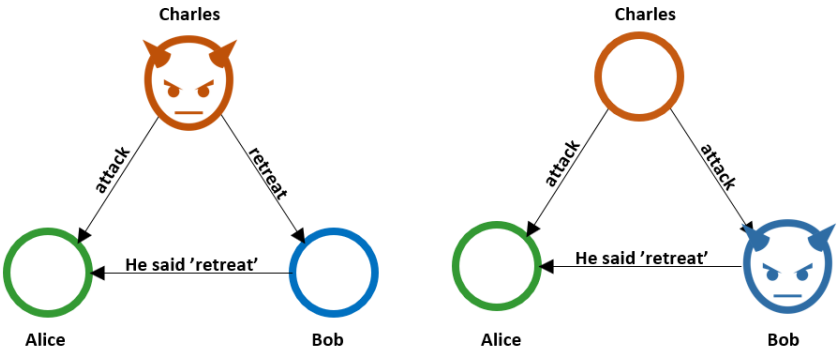

Figure 2.1: On the left, a situation where Charles is the malicious general. On the right, Bob is the malicious general. From Alice's perspective, the situations are identical, making it impossible for Alice to make a correct decision.

2.2. Consensus Algorithms

Since consensus is a fundamental problem in computer science, a tremendous amount of research has been performed in this area [2]. One of the first algorithms to offer a formal solution and safety proofs was the Paxos-family, first published in 1989 [27]. The Paxos algorithms come in two flavors: single-value and multi-value Paxos. In single-value Paxos, nodes must reach an agreement on a single value, whereas multi-value Paxos' goal is to reach agreement on multiple values and as well as on an ordering of those values. While Paxos does give guarantees on certain properties, it is impractical due to its complexity, even the practical Byzantine fault tolerance variant described in [28] comes with a communication complexity of $O(n^3)$. While this complexity is not a limiting factor when only using several nodes in, for example, distributed storage, it becomes unusable for large scale networks with large amounts of nodes. Bitcoin and Ethereum, the two largest blockchain networks currently in use, have about 10.000 and 7.000 known nodes respectively, a communication complexity of $O(n^3)$ directly renders practical Byzantine fault tolerance unfeasible [8][9].

Distributed ledgers such as Bitcoin and Ethereum employe a different method for reaching consensus; proof of work. The specific proof of work mechanism used in Bitcoin is called Nakamoto-consensus, named after its inventor. Nakamoto-consensus is essentially a lottery-based algorithm, the validator-nodes (called miners in the Bitcoin network) randomly set a field until the hash of the total block starts with a certain amount of zeroes. Since a cryptographically secure hash is designed to have a uniform distribution output, the only method for finding a valid value is brute-force. The probability of a bit being '0' is 0.5 and is independent of the other bits, meaning the difficulty exponentially increases with the number of bits ($P = 0.5^n$, where n is the number of bits). This lottery can not be easily cheated as it is physically limited by the power of the hardware. The difficulty is dynamically changed to compensate for variations in the computational power of the network and is kept to have an average of six new blocks per hour. The probability of two nodes finding a valid block at the same time is relatively low but not impossible. If this does happen, nodes may randomly pick on of the two blocks and consider this as valid, and try to find a consecutive block for this block. Since the probability of simultaneously finding consecutive blocks decreases exponentially, one chain of blocks will eventually become longer. A rule in the Bitcoin network state that the longest chain is accepted as the truth, thus the longest chain will be accepted as the truth, and all other transactions in the shorter chain (called a fork) will be regarded as invalid.

While Nakamoto consensus is proven to scale up to 10.000 miners, it has some shortcomings. Firstly, the time it takes to create a block is invariant of the number of users on the network (it always takes about 10 minutes), and since every block has a maximum number of transactions, the network's throughput is limited to seven transactions per second, regardless of the number of users. Secondly, all miners would like to be the first to find

the block and will use all of their computational capacity to find the right hash, but only one can be the first, resulting in a very wasteful and energy inefficient network. Because of these shortcomings, companies like IBM and Ripple research higher throughput and more energy-efficient methods of reaching consensus [14] [10].

The main goal of the participants of a distributed ledgers is to reach consensus on which transactions are accepted in the ledger. To determine which transactions can be accepted, we have the following requirements:

1. *Well-formedness*: Does the entry comply with the formatting specified by the network.
2. *Ordering*: To determine the current state of a ledger, the transactions must have a specific order.
3. *Validity*: based on the current state of the ledger, does the entry specify a valid action.

An invalid transaction could be a malformed transaction, or a malicious transaction trying to commit fraud by spending money or trading goods twice (known as a double-spending attack). Checking if a transaction is well-formed is close to trivial, as this can be done independently of the state of the ledger. To determine if a transaction is valid, one first needs to know the current state of this ledger. If, at any given point, it has to be possible to determine the current state, there has to be a total ordering of transactions. As different nodes should give the same verdict on the validity of a transaction, they must also agree on the same order of transactions.

2.2.1. Hyperledger Fabric

IBM's solution comes in the form of Hyperledger Fabric, and focusses on permisioned networks. Fabric uses a three-phase process for finalizing transactions. All transactions follow the process flow from the 'Endorsement phase' to the 'ordering phase', and are then finalized in the 'validation and commitment phase'.

To get a grip on the process, it is helpful to first understand the three main distinctive roles: Client, peer, and orderer. A client will create and submit a transaction, The peers possess the full state of the ledger and validate transactions, and the orderer is responsible for reaching consensus.

Figure 2.2 shows the life cycle of a transaction. During the endorsement phase, a client proposes a transaction to one or more peers (1). The peers take the proposal, along with the current state of the ledger, and simulate the transaction (2). During this step, invalid or malicious transactions can be filtered out if they conflict with the current state of the ledger (e.g., Spending funds one does not own). If the peer deems the transaction valid and legal, they send an endorsement back to the proposing client (3), essentially stating: "To the best of my ability, I have verified this transaction, and I endorse this proposal". The client now submits the endorsed transaction to the orderer, entering the ordering phase.

Up to this point, consensus has not yet been an issue. However, now the orderers have to work their magic and compile endorsed transactions into a new block. Since different sets of possibly non-overlapping peers may have endorsed different transactions, conflicting transactions may have been endorsed simultaneously. The orderer will have to find a legal combination and order of transactions and create a new block (5). After creating this new block, it will be distributed amongst the peers (6), which in turn will inform the party about the result (7).

When there is a single orderer, the orderer can easily decide to omit one of the conflicting transactions until the conflict is resolved. When multiple orderers are deployed, deciding which transaction to drop is the part where consensus becomes a problem.

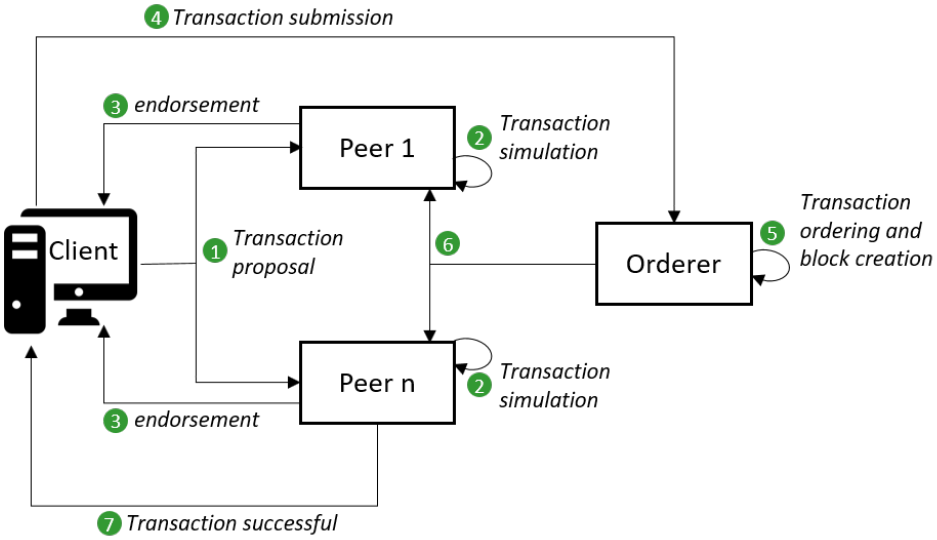

Figure 2.2: An overview of the process flow of successful transactions in Hyperledger Fabric.

During the initial development, Practical Byzantine Fault Tolerant (PBFT) was used as the consensus algorithm of choice. However, since version v1.0 they have moved away from PBFT in favor of Apache's open-source, off-the-shelve Kafka for their ordering due to performance. By doing so, they decreased the latency to $1/5^{th}$ and increased the maximum number of orderers from 16 to 26 [29]. Kafka uses a leader-follower mechanism, where a leader is elected to the ordering, and leaders learn the decided set of valid transactions, which they copy if the block is valid. If the leader crashes, a learner may start a vote to become the new leader.

A downside from moving to Kafka is that the byzantine fault-tolerant mechanism is replaced by a crash fault-tolerant algorithm. This change means that nodes are allowed to crash but will break the algorithm if processors purposely deviate from the protocol to disrupt the consensus reaching or even halt it completely.

From a business perspective allowing only 26 orderers might be worrisome. Imagine 100+ companies using a distributed ledger for their day-to-day business. From these 100+ companies, only 26 can be selected as orderers. Besides the social tension from having these 'privileged' companies, this also opens the door to abuse. The companies running orderers are the companies that decide which transactions will end up on the ledger. A malicious company could be elected as a leader and decide to block or delay a competitor's transaction for its own individual gain.

2.2.2. Ripple

Ripple aims to create a permissionless payment network. Due to this permissionless nature, nodes may enter the network at any given moment. This led the developers of Ripple's Consensus Algorithm to create a system that does not require full knowledge of all the participating nodes. They do so by requiring nodes to pick a *Unique Node List* (from now on UNL), a set of nodes they partially trust, and will only engage in the consensus algorithm with them. The reasoning being that if every node has a different Unique Node List and everyone in a UNL will reach consensus, due to overlap in UNLs they must eventually reach consensus as a whole.

Reaching consensus is done in multiple rounds. When started, the nodes have a 2-second window to take all valid transactions they are aware of and publish them in a so-called can-

didate set. The algorithm prescribes a deterministic way of sorting transactions. Therefore different nodes with the same transaction set will produce the same candidate set.

After this initial submission window, each node merges the candidate set from each node in their UNL and casts a vote on the validity of each transaction. Transactions that reach more than 50% of the votes are taken to the next round, where this process is repeated. In every round, the threshold for acceptance is increased by ten percentage points. When more than 80% of the nodes agree, the transaction is considered valid and is added to the ledger.

In the original work, it was thought that an overlap of 20% would result in a correctly functioning network [14]. However, in independent follow-up research, it was shown that the minimum overlap should be at least 40% [18]. In response, the original authors did a deep-dive and found out that the actual lower-bound was roughly 90% [19]. As a temporary solution, Ripple publishes a default UNL that should be used by all nodes, to enforce a 100% overlap. As future developments, the Ripple researchers are looking into Cobalt, the next iteration of the ripple consensus algorithm, as it requires an overlap of only 60% overlap [20].

2.2.3. Tendermint

The authors of Tendermint describe it as “a general-purpose blockchain consensus engine that can host arbitrary application states” [12].

A voting-based Byzantine fault-tolerant mechanism is used to reach consensus. An interesting feature is that the weights of the vote don’t have to be equal. This makes it possible to give parties with a larger share or higher trust more voting power. This algorithm proceeds in rounds. To make sure progress is made, the algorithm assumes a weak form of synchrony. It also assumes that per round no more than $\frac{1}{3}$ of the network experiences byzantine or crash failures.

Every round progresses in the same fashion: Proposal →pre-vote →pre-commits →Commit. A graphical overview of this process is shown in Figure 2.3.

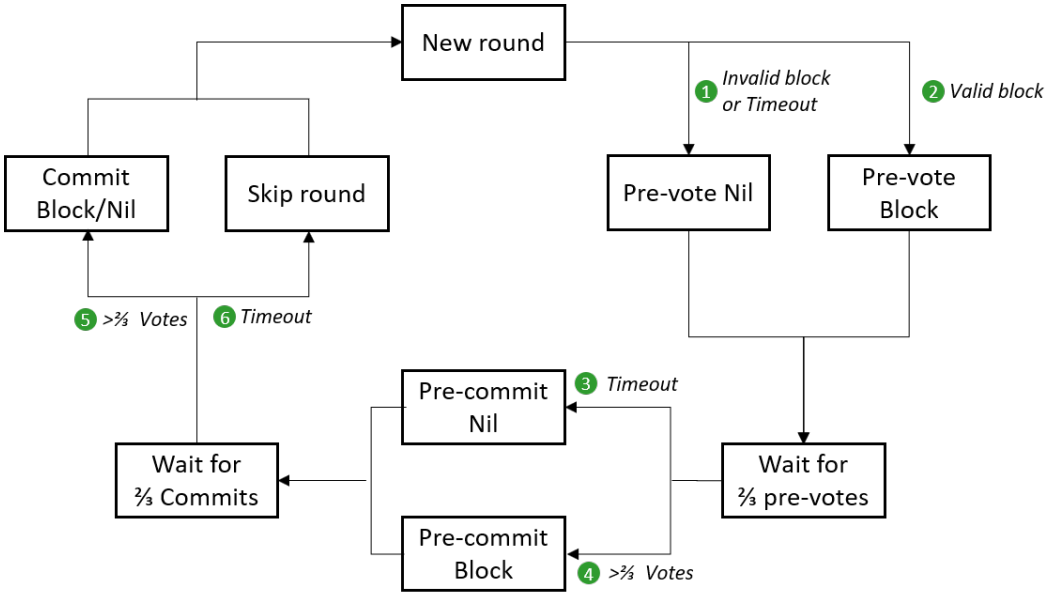

Figure 2.3: An overview of the process flow of successful transactions in Tendermint.

Every block is proposed by a single leader, and every round a new leader is selected. To reduce the overall time of the algorithm, the leader is selected using a round-robin scheme. The

newly selected leader is then expected to propose a new block with unprocessed transactions this node is aware of.

As soon as a node receives the block, they verify the validity of it, based on their current view of the ledger. If it determines that the block is valid, it will broadcast a 'pre-vote block' message (2); if no block is received or deemed invalid, it will broadcast a 'pre-vote nil' (1).

After casting its vote, each node waits until it has received at least $\frac{2}{3}$ of the votes (4), or a timeout occurs (3). If it has received at least $\frac{2}{3}$ of the votes, it will broadcast a 'pre-commit' message. Committing to what it claims to have received a majority vote. This claimed vote could either be a pre-vote for a block or a 'pre-vote Nil' (In case of an invalid block or failing proposer).

When the pre-commit is sent, each node again waits until it has received at least $\frac{2}{3}$ of the votes (5), or until a timeout occurs (6). If a timeout occurs, the round is skipped, and a new round started; A new leader is selected, and nothing is added to the ledger. If enough pre-commits are received, the agreed-upon value is committed to the ledger. The value added to the ledger does not necessarily have to be a block, but in case of an unsuccessful round can also be all 'pre-vote nil' messages or all 'pre-commit nil' messages for accounting purposes.

The reason to first have a pre-vote, and then vote again is due to the potential of byzantine leaders, who might propose different blocks to different nodes. The purpose of the pre-vote is essentially reaching an agreement on which block the nodes will try to reach an agreement.

[11] claims to have made significant performance improvement compared to PBFT. However, in initial performance testing, it was shown that only considering crash-failures, the performance dropped by about 50%. Another difficulty is the tuning of the timeout parameters; A too low value would result in unnecessary round skipping due to variance in network latency, where a too high value would result in slow recovery.

2.2.4. HoneybadgerBFT

HoneybadgerBFT [13] is a byzantine fault-tolerant consensus algorithm that does not rely on any (weak) synchrony assumptions. By being a completely asynchronous algorithm, it also does not rely on hard-to-tune time-out parameters, such as in PBFT and Tendermint. The authors claim that they have reduced the complexity by a factor of $O(n^2)$ compared to PBFT, bringing it down to a $O(n)$. It does so by making improvements on two distinct parts of PBFT.

The first step is to avoid the bandwidth bottleneck created by the reliable broadcast primitives. PBFT relies heavily on reliable broadcasts and uses a primitive first introduced in [15], which has a communication complexity blow-up of $O(n^2)$, making consensus reaching incredibly wasteful in terms of bandwidth consumption.

In HoneybadgerBFT, the atomic broadcast is replaced by a novel technique presented in [16]. In this protocol the sender uses a $(k, n)erasure - code$ [1] to split a message of size m into n pieces (n being the number of nodes), and sets k to be the maximum number of malicious nodes. This does require that the total network size to be known at any given time, as this is required to calculate the tolerable number of malicious nodes. The sender of the message will then send a different piece (of size $\frac{m}{n}$) and a hash of the original message to each node, reducing the communication complexity from $O(n^2)$ to $O(n)$ for the first step. The nodes now execute an echo-phase similar to [15]. Because nodes receive differents parts from every node, the original message can be reconstructed. By definition, no more than k nodes can fail, so no more than k message will be lost. While the message complexity is still $O(n^2)$, the reduced message size of $\frac{1}{n}$ results in a communication complexity of only $O(n)$. The author's key insight is that the execution of PBFT is limited by the bandwidth, not round-trip delay

[1] An error-correcting codes that can tolerate up to k-out-of-n corrupted words.

time.

The second improvement is made by preventing redundant transactions. A client is likely to send its desired transaction to multiple servers to reduce the chance of it getting lost, which the servers will then broadcast to the other servers. However, if n nodes each broadcast the same m transactions, the overhead of communication is a factor n due to duplicates. HoneybadgerBFT solves this problem by instead of sending all transaction, each node randomly selects a subset of its transactions. The larger the pool to select from, the smaller the chance of duplicates. This, however, opens the door to selective prevention of transactions. An adversary may delay messages for an arbitrary time and can, therefore, choose to prevent transactions it does not like. To prevent this attack, HoneybadgerBTF uses threshold-encryption to first encrypt the transactions before broadcasting. The messages can then only be decrypted after they are all combined, making it impossible for an adversary to pinpoint which exact messages contain the transaction to be censored.

The authors claim that this prevention of duplicate transactions results in another $O(n)$ improvement on the algorithm. This improvement only holds if the probability of accidentally selecting the same transaction is low. However, due to the birthday problem, this probability increases exponentially with the size of the network. The authors realize this and therefore state that the buffer size should be $\Omega(\lambda n^2 log(n))$(where n is the number of nodes, and λ is the security parameter) to ensure optimal performance.

According to [13], the algorithm's final communication complexity is only $O(n)$. The problem with choosing large buffers is that they require more transactions to fill up. For a constant influx of transactions in a growing network of nodes, the buffer size should quadratically increase with respect to the network. However, as the influx remains the same, either the complexity during transaction selection increases from $O(n)$ back towards $O(n^2)$ due to duplicate transactions or the nodes must increase the time (also quadratically) between block proposal to ensure optimal complexity but significantly increasing the latency.

2.2.5. Algorand

Algorand claims to be "a permissionless, pure proof-of-stake blockchain that delivers decentralization, scalability, and security". In [33], the authors first describe how they create a permissioned proof-of-stake, and later on suggest how this can be extended to a permissionless network.

Algorand's approach is very similar to any leader based consensus algorithm. Just as PBFT and Tendermint, it follows the same steps (Proposal→pre-commit →commit) but uses different terminology.

To select a leader, Algorand uses a lottery-based system by using a signature scheme as the basis for a verifiable random function, and then interpreting the result as an integer value. The idea behind this is that the node with the highest value (called priority in [33]) will be the elected as leader. To minimize the number of potential leaders, the paper determines a minimum priority, based on the number of shares in the network. If a node's priority is higher than this threshold, it will propose a block; otherwise, it halts its execution of the algorithm. The threshold proposed in the paper should result in at least one, but no more than 70 potentially leaders regardless of the network size [33].

To reach consensus on the proposed blocks, a second lottery is held. This time not to elect a leader, but to drastically reduce the number of participants in the agreement phase. The threshold for this lottery should be set in such a way that between 2000 to 4000 nodes win the lottery. These nodes are allowed to participate in the agreement protocol and will broadcast proof of their lottery outcome to the network. By the time each node has broadcasted this, they have also received the proposed blocks, as both lotteries can be held at the same time.

Where pBFT and Tendermint use a prepare/pre-vote to deal with a malicious leader (where they send a different block to everyone), Algorand uses a process they call *reduction*. During the reduction step, each node selects the block with the highest priority (they may have received multiply proposals) and broadcasts its hash as a vote to the network. Each node will tally the received votes, and if a single hash has received more than $\frac{2}{3}$ of the votes, they will broadcast this hash, if no majority exists the hash of an empty block is broadcasted.

After the reduction to a single value (block hash or empty hash), the nodes enter a binary agreement protocol. This binary consensus can produce two kinds of outcome: *Tentative* and *Final*. If a single node reaches final consensus, any other node that reaches final or tentative consensus in the same round must agree on the same block value. To reach final consensus, a proposal must at least receive $\frac{3}{4}$ of the votes. If the highest priority proposer was honest, the algorithm will always execute in the first two steps of the algorithm and produce final consensus. If no consensus was reached after step 2, a common coin is flipped to choose between the block hash or empty hash. This flip has a probability of $\frac{1}{3}$ of solving consensus, so the expected number of steps will be $3 \times 3\ steps$.

When the network is only weakly synchronous, different nodes can reach tentative consensus on different blocks, resulting in one or more forks. When the network is partitioned in groups with different views, no progress can be made since nodes do not count votes of nodes that have a different view. When this occurs, a recovery process must be started, which is done using the same agreement protocol but now to reach an agreement on which fork to follow. However, since consensus can only be reached in a strongly synchronous network, it can not recover from a fork until the network returns to a strongly synchronous state.

The algorithm can, however, only guarantee that the recovery time is limited to s if, in the past period of length b, there existed a period of strong synchrony longer than s. If this condition is not met, then Algorand can not recover; thus, it is required to run the recovery process preventively periodically.

The proposed method of moving the system from a permissioned to a permissionless system also raises some concerns. The authors propose that the probability of winning the leader and committee lottery should be proportional to the wealth of the party, and then assume that less than $\frac{1}{3}$ of the wealth subsides with malicious parties. However, looking at the wealth distribution, one may argue that this is unfair. In [17], it is shown that 10% of the US's wealth is owned by 0.01%. If this wealth distribution would directly apply to Algorand, it would result in a significant concentration of voting power for a selected few. When looking at the top 1% of the world, it is shown they own more than 40%, a large enough portion to (selectively) prevent consensus making. The top 10% even owns around 70% of all the wealth, resulting in enough combined voting power to successfully propose malicious transactions.

2.3. Trustchain

Trustchain is a distributed pair-wise ledger developed at the Delft University of Technology. It made its first occurrence in Tribler, an anonymous peer-to-peer file-sharing application. With a user base of 200.000 unique users and 50.000 daily concurrent users, it is a valuable scientific sandbox for research and development in the field of distributed systems.

One of the research directions within Tribler is cooperation within distributed systems; How to prevent users from free-riding, the act of only downloading from the network without giving back to the network? One of the outcomes came in the form of a completely new type of ledger, to act as an accounting mechanism for consumed and produced bandwidth; Trustchain.

Trustchain distinguishes itself from traditional DLTs by not having a single ledger containing all transactions but having a ledger for every unique user. Every block is created by multiple parties, and the parties involved in the transaction will always be among the parties creating

the block. When two or more parties engage in a transaction, they jointly create the block. The newly created block contains not only contains a signature of all involved parties, but also contains a hash pointer for each party. The jointly created block is appended to the chain of all involved parties and irrevocably entangles ledgers of all involved parties.

In a traditional blockchain, each block only points to a single previous block, linking all transactions together in a single global ledger. A simplified example of a traditional blockchain is shown in Figure 2.4, for simplicity each block only creates a single transaction. The first block represents a transaction from *p1* to *p2*, the second from *p2* to *p3*, and so forth.

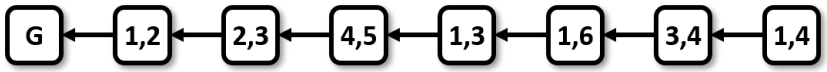

Figure 2.4: A graphical depiction of a traditional blockchain, each square represents a block, and each arrow depicts a hash-pointer. For simplicity, every block has 1 Transaction, '1,2' represents a transaction between p1 and p2, where '1,x' represents a transaction between p1 and an external party to not over-complicate the image.

In Figure 2.5, the same transactions displayed but now using the Trustchain data structure. Instead of using a single field pointing to the hash of a previous block, each block now contains two fields and thus points to two blocks. In a normal blockchain, the tamper-resistance comes from the fact that a change in a single block affects all following blocks, and in most cases can only be successfully circumvented by having more than 51% of stake or processing power within a network. In Trustchain, this mechanism is very similar; If for example, $p1$ and $p2$ collude and tamper with the transaction in the first block, all following transactions are invalidated. If they want to be successful in this scheme, they would need to convince all parties that have a path following the hash-pointers back to this transaction.

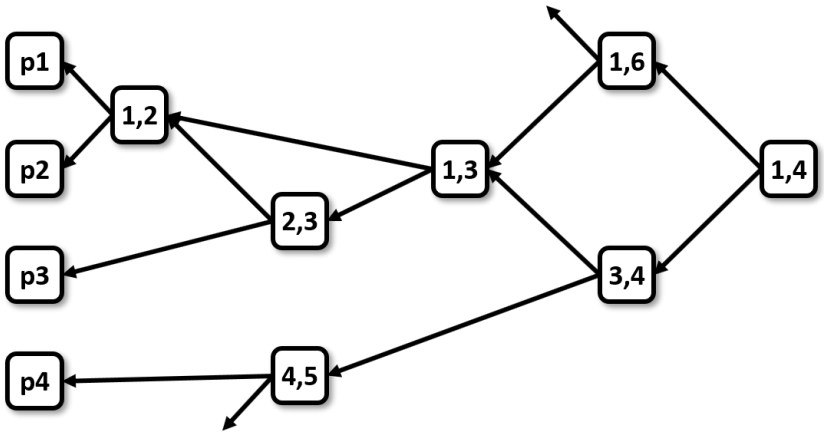

Figure 2.5: Here the same transactions are shown, but now in a Trustchain data structure. Each square depicts a block, the blocks containing p1 to p4 are the genesis blocks, every arrow is a hash-pointer, the unconnected arrows represent interactions with parties other than p1, p2, p3, p4 (these parties are omitted for clarity). Every block points to two other blocks.

Since each block points to a previous block, each block effectively is a record of an happened-before relation as described by Lamport in [21]. Due to these pointers, and the transitivity of the happened-before relationship (denoted: $\rightarrow$), each ledger is self-contained, and consistent with respect to the happened before relation. Since every party involved with the transaction is required to co-create the block, no transaction that affects a party that is not recorded on that party's ledger can exist.

From the transactions in Figure 2.5, it can be reasoned that causal ordering of transactions is already sufficient for correct execution of these transactions. To execute transaction $(3,4)$ only the order $(2,3) \rightarrow (1,3)$, $(1,2) \rightarrow (2,3)$ is relevant to asses the validity. The exact timing of $(4,x)$ with respect to $(1,2)$, $(2,3)$, and $(1,3)$ is irrelevant for the validity of $(3,4)$ as long as no transaction y exists such that $(4,5) \rightarrow y$.

By definition if $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$, events a and b are logically concurrent and can therefore be executed in parallel without affecting each other. As a result, multiple parties (that do not depend on, or interact with each other) can transact concurrently with each other, potentially leading to a highly scalable distributed ledger. By having this inherent causal transaction ordering, the necessity for total order can be dropped.

When looking at the three requirements for a valid transaction (well-formedness, ordering, and legality), the remaining reason for global consensus would only be to prevent illegal (potentially malicious) transactions. When illegal transactions can be prevented without global consensus, the requirement for a single data structure containing all transactions and a majority vote per transaction can be dropped.

Currently, there is no mechanism deployed to prevent malicious transactions in Tribler since there is little to no incentive of fraud as the bandwidth does not represent any real-world value. However, several mechanisms have been designed to at least be able to detect fraud as ways to deter potential fraudsters. The research group behind Tribler is also tirelessly working to develop reliable generic trust and reputation mechanisms, as a way to promote good behavior and cooperation.

2.4. Problem Description

Without questioning the capabilities of the researchers at companies such as IBM and Ripple on developing new consensus algorithms, one might wonder if global consensus is merely a possible solution, but not necessarily the optimal solution. The goal of global consensus in distributed ledgers is to prevent invalid transactions from being accepted into the ledger. However, it comes at the price of poor scalability in either the throughput or the number of validators. All in all, the use of global consensus as a tool to create a secure distributed ledger might not be worth the problems that come with it.

With promising new types of ledgers such as Trustchain, a truly scalable ledger might finally be in reach. However, they only offer fraud detection, no fraud prevention. Fraud detection and reputation mechanisms alone will not make these very interesting for fields where fraud is unacceptable, such as the financial industry, even with a detection rate of 100%. If a ledger only employs fraud detection, there still is a chance that a party will fall victim to fraud. While it will be eventually detected, it might take too long, or is to difficult to resolve automatically. If fraud is prevented, no party runs the risk of being a victim of fraud and therefore requires no trust in the counter-party.

In this thesis I explore the possibility of creating a secure distributed ledger while not relying on the use of global consensus. This leads to the question being answered in this thesis:

"How to achieve a truly scalable secure distributed ledger without global consensus?"

This research question covers multiple aspects that all need to be addressed in order to answer the question. 'Truly scalable' has multiple meanings in this context:

1. *Scalable in the number of validator nodes*: There should not be a limit on the number of validators that are supported by the system, neither should the throughput decrease when more validators are added.
2. *Scalable in the number clients*: There should not be a limit on the number of clients, neither should the throughput decrease when more clients are added.
3. *Scalable in transaction throughput*: There should not be a theoretical upper bound due to algorithmic constraints.

The security aspect in the context of a distributed ledger comes in the form of being insusceptible to malicious transactions, such as double spending, illegitimate creation and destruction of tokens or assets, creating forks, or other transactions that violate the system's policy. Security also comes in the form of non-repudiation; a party should not be able to create a transaction and then later on convincingly claim they did not partake in this transaction.

Having a system work without global consensus does not only imply that the validity of a transaction requires a majority vote. It also implies that nodes should be able to decide on the validity of this transaction by using partial knowledge. As complete knowledge of the network requires the existence of a single truth, which in turn implicitly demands all the nodes to agree on this single truth. Therefore the solution must not rely on the existence of a complete view of the transactions or a single truth.

3

Fair Witness Selection Protocol

This chapter will describe the key contribution of this thesis: The 'Fair witness selection protocol': The proposed solution that ensures the security of our distributed ledger. Informally, the core concept of this solution is to have witnesses sign off on valid transactions. As this algorithm is an integral part of a distributed ledger, it must comply with the three main goals of this work: Security, scalability, and no global consensus.

It must be capable of preventing malicious transactions, such as double-spending as well as preventing forks of a ledger, but may not rely on global knowledge or reverting to gossip about the ledger.

Secondly, the number of witnesses should be sub-$O(n)$ with respect to the network size. This upper bound is critical when designing for scalability; Even if a single message per witness is required, if the number of witnesses increases with $O(n)$, so will the message complexity and therefor creating a bottleneck.

Lastly, nodes should not have the ability to influence the witness selection as this would give malicious actors the ability to select malicious witnesses only. On the other hand, the witness selection should be verifiable; for any given transaction, it should be verifiable that the involved witnesses are selected according to the rules.

3.1. Formalization

We begin the description of the system by defining the following terms:

- **Node** A node is a single entity running the Trustchain network. Every node within the network must be able to aid in the ratification of transactions (I.e., be a witness). A node is allowed to engage in a transaction with another node and propose it to the network.
- **Transaction** An agreement between two or more parties, including a publicly verifiable signature from every party.
- **Witnesses** A node that is not involved in the transaction but is chosen to oversee the transaction and approve it in case of a valid transaction.
- **Block** A block is a data structure containing the transaction, the witnesses' IDs and the corresponding signatures, finalized by a hash.

- **Chain** A chain is an ordered set of blocks, where each block contains a pointer to a previous block (except for the genesis block, which represents the start of a chain). The chain of a specific node comprises of the ordered set of all blocks co-created by that party is meant.

3.2. Adversarial and System Model

The Fair Witness Selection Protocol (from now on FWSP) makes certain assumptions about the system. We assume that all the nodes have a private and authenticated channel for communication (for example, through the use of public-key cryptography). A weak sense of synchronization is implied trough to use of messaging time-outs. This is done to overcome the deterministic termination impossibility in asynchronous systems, as described by Fischer, Lynch, and Paterson[32]. We assume that every node has contact information and the signature verification key for every other node (or means to acquire this), be it through a predetermined look-up table or a byzantine-fault tolerant DHT[31]. As proven in [26], there are no solutions for consensus algorithms with more than $\lceil \frac{n}{3} \rceil - 1$ malicious nodes. Therefore, we assume this as the upper-bound on the malicious fraction of the network. It is assumed that unique identifiers of nodes are in the same domain as the hash function (If this is not the case, the unique identifiers can easily be mapped to the same domain by applying the hash function to each identifier).

We use the term *nonfaulty* to refer to nodes in the network that behave honestly and without error. A *malicious* nodes may deviate from the algorithm (Byzantine errors) and randomly send or refuse to send messages. It is assumed that:

- all nonfaulty nodes only propose valid transactions, responds timely (i.e., before message time-out), and will sign every valid transaction.
- Crashing nodes may stop sending messages at any time and do not resume at a later point. Messages are either send entirely or not send at all.
- Byzantine nodes are allowed to start, stop, or deviate from the algorithm at any give time. They may send, or refrain from sending, any message with any arbitrary content.

Byzantine nodes are considered to be a subset of faulty nodes. If, for example, it is stated that $\lceil \frac{n}{3} \rceil$ is faulty and $\lceil \frac{n}{5} \rceil$ is malicious, these malicious nodes are considered to be byzantine and in the set of faulty nodes, whereas the remaining nodes are considered to be crashing node.

3.3. Fair Witness Selection Protocol

The core concept of the fair witness selection protocol is that every block is witnessed by a number of nodes. Only with enough witness signatures a block is considered valid.

Now the question arises, how to pick witnesses in a random but verifiable way. Random, as no party should be able to forge a transaction in such a way the can selected a set of witnesses. But Verifiable, in the sense that anyone in the network should be able to verify that those witnesses were correctly selected.

The design goals are as follows:

- Every valid transaction must be accepted in finite time.
- An invalid transaction will be accepted with a negligible probability.

3.3.1. Definition of the Algorithm

Let N be the set of all nodes, W the set of selected witnesses, M the set of all malicious nodes, H the set of all honest nodes, and k the minimum number of required signatures. Such that $|N| = |H| + |M|$, $|M| \leq \lceil \frac{|N|}{3} \rceil - 1$, and $k = \lfloor \frac{2|W|}{3} \rfloor + 1$. We furthermore specify s to be the minimum size for the witness set, this number vastly impacts security (see Section 3.3.4 for suggested values). Let $V(t, b) \rightarrow \{true,\ false\}$ be a deterministic function that takes transaction t and an array of blocks b, and outputs $true$ if and only if t is valid. Let $H(\{0,1\}^r) \rightarrow \{0,1\}^n$ be a cryptographically secure, uniformly distributed hash, that takes a message of any length with an output of length n.

Let $Sign(\{0,1\}^r, sk) \rightarrow \{0,1\}^n$ be a signing function of a secure signature scheme, and $Ver(\{0,1\}^n, pk) \rightarrow \{true,\ false\}$ the corresponding verification function, where sk is the private key and pk is the public key and n is the signature length in bits.

The core strength of FWSP lies in the witness selection. The i^{th} witness is selected through:

$$w_i = \sup\{z \in N : z_{ID} \leq H(t||i)\} \tag{3.1}$$

An honest node can create a valid block by selecting its witnesses using Equation 3.1 and request those witnesses to sign the transaction. Due to the uniform distribution of the hash function, nodes are essentially selected uniformly at random. Furthermore, the pre-image resistance of the hash ensures that its difficult to find a transaction that maps to a specific witness.

A larger minimal witnesses set results in a higher security level. In Section 3.3.2, a mathematical proof of this security parameter, along with a minimum witness set for a variety of assumptions and requirements is given. A block is considered valid if the transaction is signed by at least k witnesses.

When a malicious actor tries to pass an inadmissible transaction, a trivial attack would be to keep increasing i until enough malicious witnesses are found. To combat this also the i used for determining the witness is included, along with the ID and a valid signature. Since k, the number of required signatures grows linearly with the witness, the adversary decreases his probability of succeeding exponentially.

The algorithm can be separated into three distinct functions: Block creation, block signing, and block verification. A formal description of each of these functions can be found in Algorithm 1, Algorithm 2, and Algorithm 3, respectively.

Block creation The creation of the block starts by verifying the validity of the transaction. If the transaction is valid, the node calculates the hash of the transaction with a 0 appended and selects the node with the smallest ID larger than or equal to the hash. The selected witness sends the transaction and the corresponding i, 0 in the initial case. The node repeats this process with an increasing value for i up to s, instead of 0.

When the requesting party receives more than $(k - s)$ ⊥s or a request times-out, it extends the witness group by increasing i by one, and selecting the witness according to Equation 3.1 once again. This process continues until the node has acquired enough (more than k) valid signatures. When the threshold is reached, the transaction is finalized by concatenating the hash of the transaction, and signatures to the block.

Block signing On receiving a signature request, the receiving node first verifies that the transaction is valid, and it indeed is the required witness. If this is not the case, the node replies with ⊥; otherwise, the node appends the given i to the transaction and signs it. By

appending the i, the node prevents any malicious actor from making false claims about the number of nodes selected as witnesses by modifying i after having received the signatures. The node's ID is concatenated with signatures, and i, and is sent back to the requesting party.

Block verification Block verification is a non-interactive process. The verification is straight forward and is started by checking if the number of signatures is equal to or greater than the minimum required number of signatures. Next, the largest witness set such that the number of supplied signatures $\geq \lfloor \frac{2|W|}{3} \rfloor + 1$ still holds is calculated. Then, it is checked that none of the supplied values of i exceed the largest acceptable witness set size. For each witness, it is verified that they were selected according to the protocol and that the signature is valid. If all of these tests pass, the block is valid, otherwise the block is considered invalid.

Algorithm 1: createBlock(transaction)

```
t ← transaction, validSignatures ← ∅
if V(t) then
    for i ← 0 to s do
        w ← sup{z ∈ N : z ≤ H(t||i)}
        send requestSignature(w, t, i)
        W ← W ∪ w
    while |validSignatures| < k do
        if requestSignature() time-out then
            i ← i + 1
            w ← sup{z ∈ N : z ≤ H(t||i)}
            W ← W ∪ {w}
            send requestSignature(w, t, i)
    block ← t
    foreach s ∈ validSignatures do
        block ← block||s
    return block||H(block)
```

Algorithm 2: on receive requestSignature(t, i)

```
t ← transaction, p ← nodeID
w ← sup{z ∈ N : z ≤ H(t||i)}
if V(t) and w = p then
    s ← Sign(t||p||i, sk)
    reply p||s||i
else
    reply ⊥
```

Algorithm 3: verifyBlock(block)

```
t, signatures, hash ← interpret(block)
if |signatures| < s ∨ hash ≠ H(block) then
    return ⊥
w_max ← ⌈|signatures| × 3/2⌉ − 1
if {∃i ∈ signatures : i > w_max} then
    return ⊥
foreach e ∈ signatures do
    w, Sign, i ← interpret(e)
    w_expected ← sup{z ∈ N : z ≤ H(t||i)}
    if not (w = W_expected and Ver(Sign, w_pk)) then
        return ⊥
```

3.3.2. Correctness

An invalid transaction will only be accepted with a negligible probability For an actor to have a malicious transaction accepted, at least k byzantine nodes in the witness set are required. It is assumed that all malicious nodes collude, as this increased the effectivity of the attack. The probability of a portion of a subset being byzantine, when drawn at random from a finite population, can be modeled with the hypergeometric distribution.

Let X be a random variable corresponding to the number of byzantine actors in the witness set. The probability mass function of X is then given by:

$$p_X(k) = Pr(X = k) = \frac{\binom{|M|}{k}\binom{|N|-|M|}{|W|-k}}{\binom{|N|}{|W|}}$$

where

- N the set of all nodes,
- M is the set of all byzantine nodes,
- W the set of all selected witnesses,
- k the number of malicious nodes in the witness set,
- $\binom{a}{b}$ is a binomial coefficient.

Not only k malicious signatures pass an invalid transaction, but any number greater than k will also succeed. The probability of a malicious transaction succeeding for a given witness set becomes:

$$Pr(X \geq k) = \sum_{i=k}^{|W|} \frac{\binom{|M|}{i}\binom{|N|-|M|}{|W|-i}}{\binom{|N|}{|W|}}$$

since a node is allowed to increase the witness set if a transaction did not succeed, the probability becomes:

$$Pr(X \geq k) = \sum_{j=|W|}^{|N|} \sum_{i=k}^{j} \frac{\binom{|M|}{i}\binom{|N|-|M|}{j-i}}{\binom{|N|}{j}} \tag{3.2}$$

due to Boole's inequality, we can state that Equation 3.2 is an upper-bound, the actual probability of a malicious transaction being accepted, however, might be lower. Depending on the maximum number of malicious actors that should be tolerated, the witness set size can be chosen such that the probability matches the security parameter. Examples of parameters are given in Section 3.3.4.

3.3.3. Liveness

Every valid transaction must be accepted in finite time Intuitively, the proof goes as followed: If not enough valid signatures are received, a node is allowed to extend the witness set. Eventually, the witness set will extend to the complete network. By definition there are at most $\frac{|N|}{3} - 1$ malicious nodes, and a transaction is valid if at least $\frac{2}{3}$ of the witnesses sign it. Therefore a transaction will always be valid if the whole network is involved.

In accordance with the algorithm, if an honest transaction receives less than k valid signatures (due to malicious behavior or due to faulty behavior), the node is allowed to increase its witness set.

They may do so as long as required. In a worst-case scenario, the witness-set entails the complete network, meaning $W = N$.

A block is valid if:

$$v \geq \lfloor \frac{2|W|}{3} \rfloor + 1$$

where v is the number of valid signatures. By definition honest nodes always sign valid transactions, thus $v = |H|$, which gives:

$$|H| \geq \lfloor \frac{2|W|}{3} \rfloor + 1$$

since $|N| = |H| + |M|$ and $|M| \leq \lceil \frac{|N|}{3} \rceil - 1$, it holds that:

$$|H| \geq \lfloor \frac{2|N|}{3} \rfloor + 1$$

therefore a block is valid if

$$\lfloor \frac{2|N|}{3} \rfloor + 1 \geq \lfloor \frac{2|W|}{3} \rfloor + 1$$

if $W = N$, the equation always holds, therefor the block will always be valid. □

Since signing and creating transactions are two separate procedures that have no dependencies on each other, other transactions can be witnessed during transactions, transactions can be started while signing, and two signature requests can be fulfilled concurrently.

3.3.4. Security

The minimum required number of witnesses relies heavily on the required security of the protocol. If a single witness would be allowed, the probability of a malicious transaction succeeding is equal to $\frac{1}{3}$. A larger witness group results in a lower probability, the exact number of witnesses depends on the security parameters and the assumed number of byzantine nodes in the network.

Equation 3.2 can be used to calculate the an upper-bound on the probability of a malicious transaction being accepted. Using this equation, it is possible to determine the minimal witness-set size for different probabilities of success. In Figure 3.1, the minimum witness set size for four different probabilities as a function of the network size is shown. The malicious nodes make up $1/3$ of the entire network. The minimum witness set size is given for $< 1\%$, $< 0.1\%$, $< 0.01\%$, and $< 0.001\%$ chance of malicious transactions succeeding.

In Figure 3.2, the minimum number of required witnesses is given, where the X-axis is the fraction of the byzantine nodes in the network, and the y-axis is the minimum number of required witnesses. In case the required number of witnesses is larger than the network size, the network size should be chosen, as this will result in a probability of 0 (See Section 3.3.2 for proof).

While designing, system engineers can provision the minimum witness set size on the portion of byzantine nodes, not crash-faulty nodes. Since nodes are allowed to extend their witness set, the minimum size of the witness set has no impact on the crash-fault tolerance. In cases where crashing node are more likely to occur than byzantine behavior, witness sets will be smaller than those based on the larger fraction of crashing nodes instead of the larger fraction of crashing nodes.

While $< 0.001\%$ for a malicious transaction to succeed is a small probability, it is still far from negligible. However, all the probabilities mentioned only involve a single transaction being verified.

The overall security can be vastly increased by requiring that not only the current transaction should be verified, but also a given number of former transactions. In practice, this comes down to not only sending a single block to the witness for validation of the current transaction, but also send the last h blocks for both determining the previous-hash pointers and verifying the validity of previous transactions.

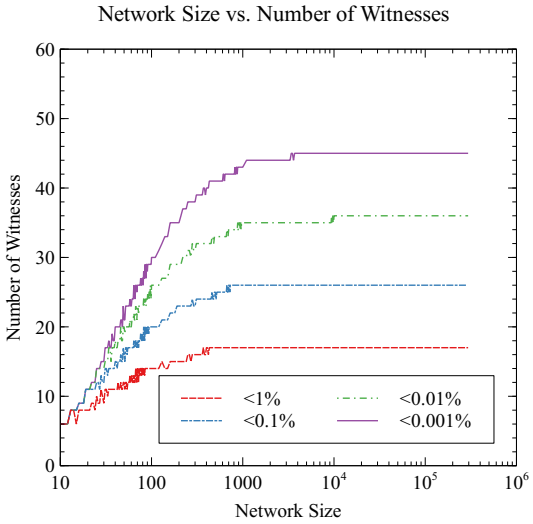

Figure 3.1: The minimum number of witnesses as a function of the network size. The malicious nodes make up 1/3 of the entire network. The number of required witnesses are given for $< 1\%$, $< 0.1\%$, $< 0.01\%$, and $< 0.001\%$ chance of malicious transactions succeeding.

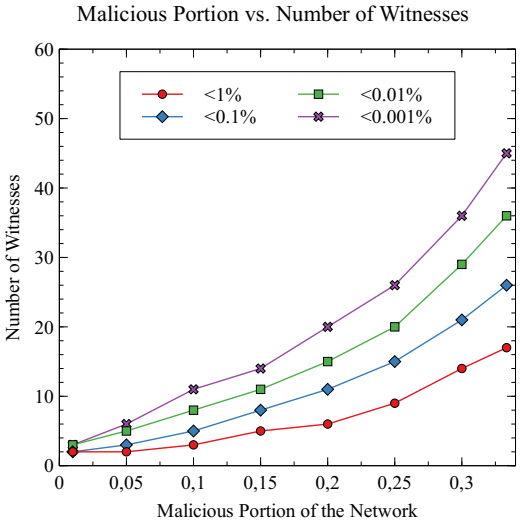

Figure 3.2: Minimum number of witnesses as a function of the byzantine portion of the total network. The number of required witnesses are given for $< 1\%$, $< 0.1\%$, $< 0.01\%$, and $< 0.001\%$ chance of malicious transactions succeeding.

A malicious party that successfully performed an invalid transaction will not be able to create a new block if they do not have a majority of malicious witnesses as well. If this malicious party would ever like to reap the seeds of his crime, he has to hide the malicious transaction behind h honest transactions, which can only be signed by a witness set consisting of a majority of malicious nodes.

Take the example given in Figure 3.3, here $tx0$ is the malicious transactions, and the network demands that witnesses verify the last five blocks. If the node would ever want to perform transactions involving non-malicious peers, the malicious transaction should be succeeded by at least five valid transactions. However, these valid transactions will not be signed by honest witnesses as they would be required the last five blocks, which include an invalid transaction. This means that the only way to get away with this transaction is to have $tx0$ until $tx5$ all be witnessed by malicious witness sets. Since the witnesses are drawn from an independent identical distribution, the probability p becomes p^6. If we assume an initial probability of $< 0.001\%$, this gets increased to $< 1 \times 10^{-18}\%$.

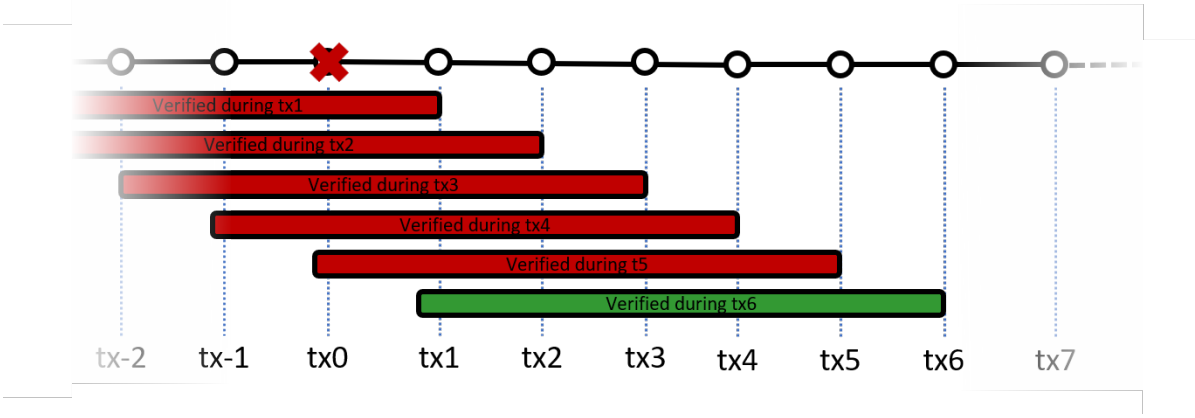

Figure 3.3: Graphical representation of a malicious transaction being followed by valid transactions. In this example, $tx0$ is the malicious transaction, $h = 5$ a red bar indicates a verification that would fail, where a green bar indicates a success.

This case can be generalized to h blocks with an initial probability of p. If the probability

of performing a malicious transaction is p, and h valid transactions need to be signed by a dishonest witness set, the probability of successfully performing an attack is:

$$P = p \times p^h = p^{h+1}$$

The National Institute for Standards and Technologies, states the highest approved security strengths for federal applications is 256-bits security[35]. For an encryption system to be considered secure, there must not exist a method of breaking the encryption with a probability of succeeding higher than guessing the private key. For a 256-bit long private key, having 2^{256} possible keys, the change of guessing the private key is $\frac{1}{2^{256}}$.

Using these probabilities, we can determine the combination of minimal witness-set size and the required number of previous transactions to guarantee 256-bit security. These parameters were also calculated for 128-bit Security, another commonly used security level.

In table 3.1, an overview of the number of previous transactions that need to be verified as a function of the witness set size and the malicious portion of the network to each 128-bit security can be seen.

$P_{Malicious} =$	33%	30%	25%	20%	15%	10%	5%	1%
$\|W\| = 10$	31	24	17	13	10	7	5	3
$\|W\| = 20$	16	12	9	7	5	4	2	1
$\|W\| = 30$	11	8	6	5	3	2	1	1
$\|W\| = 40$	8	6	5	3	2	2	1	0
$\|W\| = 50$	6	5	4	3	2	1	1	0
$\|W\| = 60$	5	4	3	2	2	1	1	0
$\|W\| = 70$	5	4	3	2	1	1	0	0
$\|W\| = 80$	4	3	2	2	1	1	0	0
$\|W\| = 90$	3	3	2	1	1	0	0	0
$\|W\| = 100$	3	2	2	1	1	0	0	0

Table 3.1: The number of previous transactions to be checked to reach 128 Bit equivalent security. Different columns represent different portions of the network being malicious, where different rows represent different sizes of witness sets

In table 3.2 the number of previous transaction that need to be verified to reach 256 bit equivalent is given.

$P_{Malicious} =$	33%	30%	25%	20%	15%	10%	5%	1%
$\|W\| = 10$	62	49	35	27	20	15	10	6
$\|W\| = 20$	32	25	19	14	10	8	5	3
$\|W\| = 30$	22	17	13	10	7	5	3	2
$\|W\| = 40$	17	13	10	7	5	4	3	1
$\|W\| = 50$	13	11	8	6	4	3	2	1
$\|W\| = 60$	11	9	7	5	4	2	2	1
$\|W\| = 70$	10	8	6	4	3	2	1	0
$\|W\| = 80$	8	7	5	4	3	2	1	0
$\|W\| = 90$	7	6	4	3	2	1	1	0
$\|W\| = 100$	7	5	4	3	2	1	1	0

Table 3.2: The number of previous transactions to be checked to achieve 256 Bit equivalent security. Different columns represent different portions of the network being malicious, where different rows represent different sizes of witness sets

3.4. Empirical Overhead Analysis

The number of required witnesses has a significant impact on the performance of the algorithm. For a party to have their transaction validated, they are required to send each witness

a request. They then have to wait for at least two-thirds of replies to the requests. All of the messages have a length independent of the number of witnesses, resulting in a message and communication complexity that are equal to the number of witnesses.

Since the communication complexity is bounded by the witness set size, it is essential to identify a function that describes the growth of the witness set size as the network increases. An empirical approach was taken to determining the smallest function that bounds the witness set size as the network approaches infinity.

Using Equation 3.2, a minimal witness set was calculated to give a $< 0.001\%$ probability for malicious transactions to take place, under the assumption of one-third of the network is malicious. Using this data, it is empirically determined that $O(log^*(n))$ is the smallest function that bounds the minimum witness set size. In Figure 3.4, the iterated log ($log^*()$), the minimum witness set size, and other commonly occurring complexities are shown.

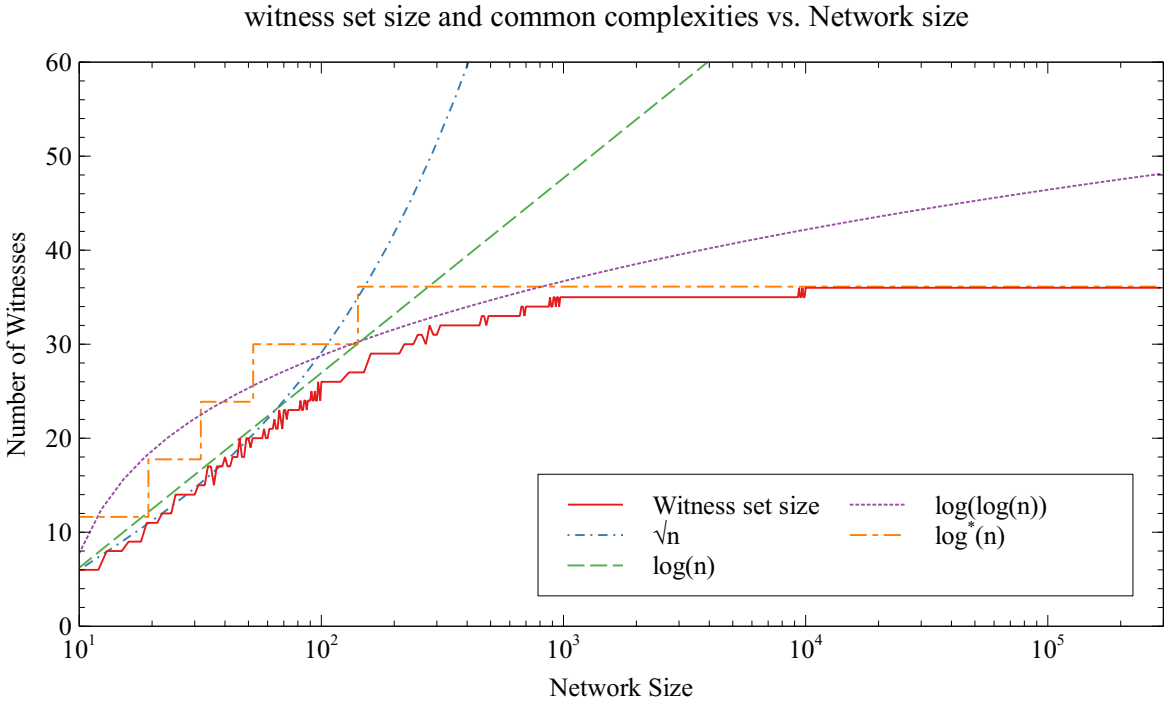

Figure 3.4: The minimum number of required witnesses plotted along commonly used complexities. The functions named in the key are the orders of magnitude of the functions; the exact coefficients are omitted.

In Figure 3.4, it can be seen that $O(\sqrt{n})$ and $O(log(n))$ are not the smallest bounding functions. While $O(log(log(n)))$ does approaches the witness set size better, it still keeps growing as the network size grows. Even when applying the logarithm multiple times (omitted from the picture for clarity), it kept diverging from the minimum witness set and the iterated log. When plotting the iterated logarithm, it can be seen that it fits the line better than $O(log(log((n))$. Therefore it is assumed that, at least within the range of up to 2×10^5 nodes, the system is best bound by $O(log^*(n))$.

3.5. Resilience Against Common Attacks

The fair witness selection protocol can be deployed against various attacks common to distributed ledgers. Based on the attack, different mechanisms can be deployed to prevent them from happening.

Inadmissible Transactions Inadmissible transactions are transactions that are well formed but are not valid with respect to the rules of the network. Examples of these would be double spending attacks, but also unauthorized creation or destruction of assets. However, as dangerous as they are to the network, protection against these attacks is rather trivial. If all requirements that constitute a valid transaction are embedded in a function used by FWSP, and a block is only accepted if it comes with the correct number of signatures, then the probability of an inadmissible transaction being accepted can become negligibly small, dependent on the FWSP parameters.

Block Withholding & Forking Block withholding attacks and forking attacks are treated the same since a forking attack essentially is a specific case of block withholding. When performing a forking attack starting from a particular block, the attacker essentially tries to hide all blocks succeeding that specific block.

To prevent block withholding, the fair witness selection protocol can be used to create a reliable way of block dissemination and replication. To do so, not the hash of the current transaction, but the hash of the previous block should be used. When combining this with the requirement that witnesses are not only supposed to attest the transactions but also store the blocks they witness, the probability of successfully hiding a block becomes even lower than that of passing an inadmissible transaction. Whereas passing an inadmissible transaction will occur only when more than two-thirds of the witness set in malicious, a single honest node in the witness set already compromises the attack. This has to do with the unforgeability and non-repudability of the signatures used to create the transactions. Thus, to successfully hide a block, all of the witnesses in the witness set must be malicious. Recall that the probability of having k malicious witnesses i a witness set is given by:

$$p_X(k) = Pr(X = k) = \frac{\binom{|M|}{k}\binom{|N|-|M|}{|W|-k}}{\binom{|N|}{|W|}}$$

Since we now require the complete witness set to be malicious, k should now equal the witness set $|W|$. Resulting in the following equation:

$$p_X(|W|) = Pr(X = |W|) = \frac{\binom{|M|}{|W|}\binom{|N|-|M|}{|W|-|W|}}{\binom{|N|}{|W|}} = \frac{\binom{|M|}{|W|}\binom{|N|-|M|}{0}}{\binom{|N|}{|W|}} = \frac{\binom{|M|}{|W|}}{\binom{|N|}{|W|}}$$

When having a network of 100.000 nodes, of which $1/3^{rd}$ is malicious, and a minimum witness set of 45 nodes will result in an inadmissible transaction having a $< 0.001\%$ chance of being accepted, where it only has a $< 3.3 \times 10^{-20}\%$ chance of successfully hiding a block.

When a node is presented a block of which it is claimed it is the last block in the chain, it can now be easily verified. Using Equation 3.1 and the presented block's hash as input, the witness set of a successive block is determined. The node can then query these witnesses on the existence of a successive block. If a successor does exist an honest node will supply the inquiring node with the bock, which acts as an indisputable proof that the other party was indeed acting malicious.

3.6. Subset Vulnerability

Which data is used as input for the hash used in the fair witness selection protocol has a great impact on the chosen witness set. If the input can be freely changed, an adversary could try enough inputs until he finds a witness group to his liking. When the probability

of picking such a group is sufficiently low (e.g., $< \frac{1}{2^{80}}$), then this brute force attack becomes infeasible. However, when relying on the technique described in Section 3.3.4 to increase security this is not sufficient.

If the probability of a single malicious transaction succeeding is $< 0.001\%$ and 15 previous transactions also need to be checked, then the security based on;

$$P = p^{n+1}$$

does not hold anymore since, on average, 100.000 hashes need to be tried per block, which is no match for a modern desktop computer. This means that an adversary can have a single transaction probability of one, thus also a total probability of one. To prevent this, two methods are proposed.

The first method only uses the public key and the sequence number as the input. This means that the nodes have no influence on which witnesses will be selected for the verification of the transaction. The downside of this approach is that a malicious node knows all future witness sets in advance.

In some cases, this might be undesirable; if this is the case, a second method can be used. This method bases the current witness set solely on the previous block. Since the signatures of the witnesses are included as well, the creating parties do not have direct control over the input of the hash function. However, in this method, additional care must be taken to ensure that all signatures given by witnesses on a previous transaction are actually used. This has to do with the fact that, by design, not all signatures are required. When, for example, a witness set of 21 is required by the network, 14 signatures are enough to let the transaction pass. If the transacting party receives all 21 valid signatures, it can select any combination of 14 or more signatures, and the block is considered valid. The amount of unique and valid subsets for this given example rises to:

$$\binom{21}{14} + \binom{21}{15} + \ldots + \binom{21}{21} = 198440$$

Even when having a 0.001% chance of a malicious majority witness set, this becomes very probable if 198.440 options are available. To prevent this, measures should be deployed to prevent a transactor from dropping signatures.

While it might be possible for a node that wants to validate a transaction to query the corresponding witnesses if they had given their signature, the protocol would lose its non-interactive block verification property and create a new attack vector. A malicious witness may refrain from signing when the transaction is made, but claim he did so when another node wants to verify the transaction to discredit the transactor.

An effective counter-measure is making the signing a two-phase interactive process. In the first phase, the transactor announces to the witnesses of his intention to make a transaction. At this point, the witnesses pledge their willingness to sign the transaction. After which all willing witnesses are added to the transaction before signing. This means that the witnesses who pledged it would sign, are part of the message over which the signatures are created. A transaction is then only valid if all of the witnesses who pledged signed the transaction is valid.

A malicious node might pledge to sign and then refrain from doing so, resulting in a failure. The transactor can, however, initiate a new round with the malicious node removed, possibly with additional witnesses. The number of rounds is then limited to the number of malicious nodes in a witness set.

4

Self-Sovereign Banking

In this chapter, a generic architecture is presented to enable truly distributed banking. By utilizing the algorithm from chapter 3, each user is the sole owner of its data, making them fully sovereign.

In a traditional payment network, all transactions flow through the bank of the bank holder, as displayed in Figure 4.1 on the left. When this transaction takes place between two accounts held at different banks, it actually has to flow to both banks and potentially even a third party, for example, clearinghouses in case of cross border transactions.

Not only does this to potential bottlenecks and puts a heavy burden on the digital infrastructure of the involved banks, but it also exposes multiple single points of failure. With €1100 billion in card payments [30] in the eurozone and almost €45 billion through the largest online payment platform [34] in the Netherlands, a successful attack on the digital infrastructure of banks can have a severe impact on a countries economy.

The goal of this proof-of-concept is to create a decentralized enterprise-oriented payment network, where parties can exchange stable-tokens (tokens which have a fixed exchange rate with respect to real-world currencies). This has the potential to have higher reliability due to the lack of single point of failures while having lower operational costs.

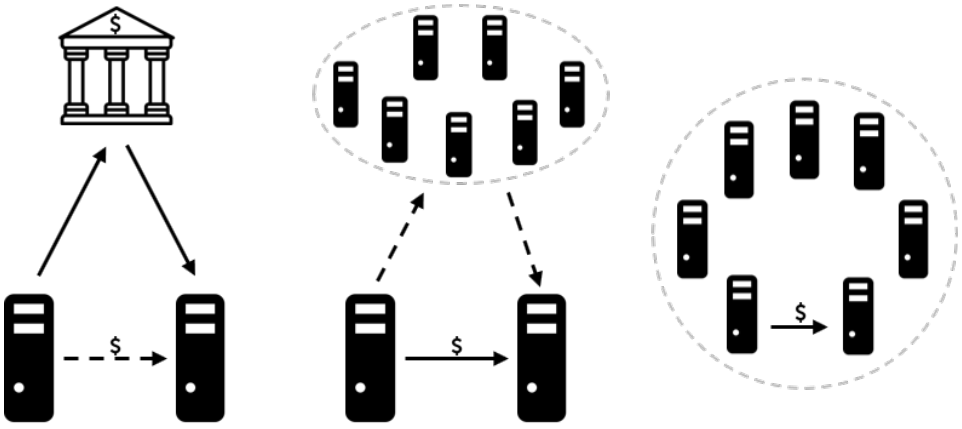

Figure 4.1: Depiction of a payment in three different payment network topologies. A dashed lined indicates the flow of information, where the solid line indicates actual flow of money.

In Figure 4.1 representation of different topologies of payment networks is shown in which Alice wants to transfer funds to Bob. Nodes in the circle are full-clients. Left) A traditional

payment network; Alice informs the bank about the transaction, which will route the money to Bob. Centre) A heterogeneous network where Alice and Bob are light-clients; Alice directly transfers Funds to Bob, but he can not use them unless verified by enough witnesses. Right) A Homogeneous network, similar to the heterogeneous network, but now Alice and Bob are required to partake in witnessing transactions.

For this design, a homogeneous network was chosen such that every node that utilizes the network also contributes to this very network. This is not necessary as it would also be possible to make a distinction between 'light-clients', that can only initiate but not witness transactions, and 'full-clients' that can both witness and initiate transactions, and 'validator-nodes' which only witness transactions.

To ease the design and enable code reusability, the software architecture is modular. The design is split up into five distinct pieces: User Interface, Business Logic, Trustchain core, and storage.

4.1. Software Architecture

The architecture of the proof-of-concept has been designed to be as modular as possible to enabled fast development and increase code reusability. Figure 4.2 depicts an overview of the modules implemented for this proof-of-concept. The application is written in Python to increase the speed of development. While Python is not known for its high performance, the impact is manageable as the application is mostly I/O bound (due to networking), and the cryptographic computations are offloaded to LibSodium, a high-performance, independently audited cryptographic library written in c++.

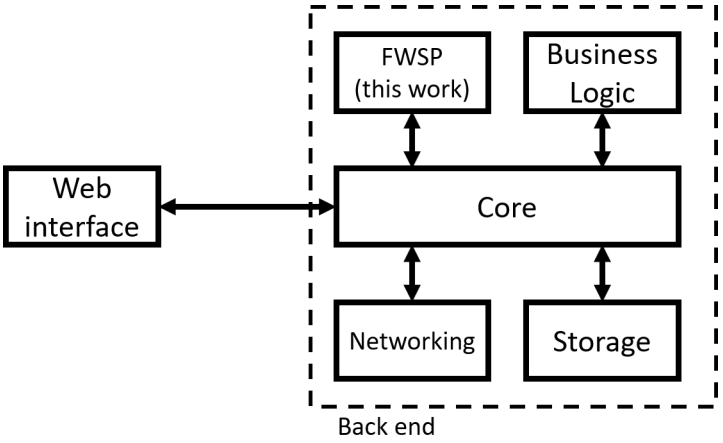

Figure 4.2: A graphical overview of the modular architecture of the proof of concept.

- The Core module contains the core functionality of the application, such as initiating transactions, accepting incoming transactions, block creation, and all cryptographic actions.
- The FWSP module is the implementation of the algorithm as described in section 3. This module contains the functionality that takes a transaction and returns the corresponding set of witnesses.
- The so-called 'Business Logic' module contains the code that describes the functionality of the application, in this case, the logic of managing and transferring funds.
- The Networking module manages the authentication of and communicating with other peers.
- The Storage module is responsible for storing the created and witnessed blocks.

- As a user-interface, it was chosen to use a web interface to give an experience similar to online banking.

The storage module is currently implemented as an in-memory storage as the demonstrator and experiments serve as a proof of concept and not as production code, and therefore will not be discussed in detail in this document. However, any storage mechanism that supports key-value stores can be used as a drop-in replacement (e.g., Redis), any other storage mechanism can be used with minimal changes. The FWSP module will also not be discussed as the inner workings are already described in chapter 3.3.1.

4.2. Networking

To enable transactions between clients, a messaging protocol has been specified. The protocol consists of three phases: the inquiry phase, 'does the partner want to transact?', the proposal phase, where a block is proposed, and the acceptance phase, where the proposal is signed by the relevant parties.

In this protocol, the initiator of the transaction, Alice, sends a message to the desired partner, Bob. In this message, she specifies the value of the transfer they want to make, along with the number of previous blocks that is required by the network rules.

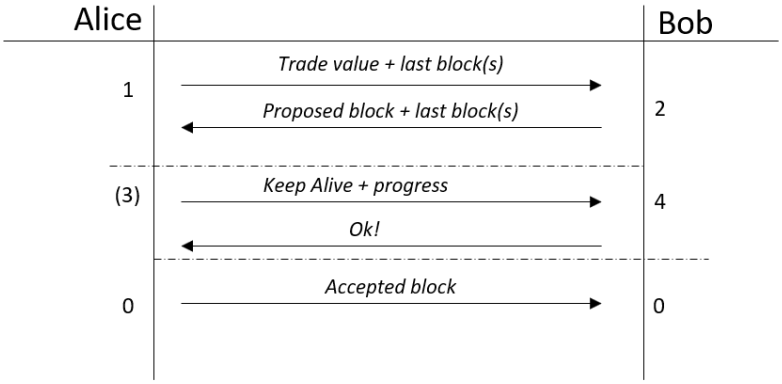

Figure 4.3: Sequence diagram of the message flow during a transaction. Here Alice is the initiator of a transaction. The messages and states in between the dashed lines are optional. The numbers are the internal state of the finite-state machine after sending the corresponding message.

If Bob is willing to accept a transaction, he will validate the previous blocks send along. The blocks, combined with Bob's own chain and the transaction value, contain all the information required to create a block. Bob uses this to propose a block, directly signs this block, and then sends the proposed block along with his required previous blocks to Alice.

Alice can then send the sign this block, and forward the blocks to the relevant witnesses for approval. Once the block is signed by at least two-thirds of the witnesses, she adds the block's hash and sends the finalized block to Bob and the relevant witnesses.

In the best-case scenario, the witnesses would reply with their signature before the connection between Alice and Bob would even be close to timing out, making the whole protocol run only requiring three messages.

It is, however, possible that a witness will take longer, due to malicious behavior or network failure. To prevent the connection from timing out, a keep-alive message can be sent. A message can be piggybacked with this keep-alive message to inform each other on the progress.

In this application, the protocol uses TCP connections. Due to the interactiveness of transact-

ing, a session-oriented protocol is more suited than message-oriented protocols such as UDP. It furthermore enables the use of TLS (Transport Layer Security) to provide authentication, privacy, and data integrity between the clients. By using server and client-side certificates, confidentiality, integrity, and authenticity can be easily added.

The proof of concept has the protocol implemented in the form of communicating finite-state machines. This allows quick detection and recovery in case of packet loss or desynchronization.

4.3. User Interface

The user interface is designed to be a web-app, to give the end-user the experience similar to online-banking. Bootstrap, a popular open-source front-end framework, is used to generate a responsive website that can be operated on both desktops and mobile devices. When navigating to the interface, the user is presented with an overview of the bank-accounts owned by the user, this can be seen in Figure 4.4. From here, the user can navigate to the transfer window, from where they can initiate a transaction with another client, shown in Figure 4.5. Alternatively, they can navigate to an overview of all previous transactions, shown in Figure 4.6.

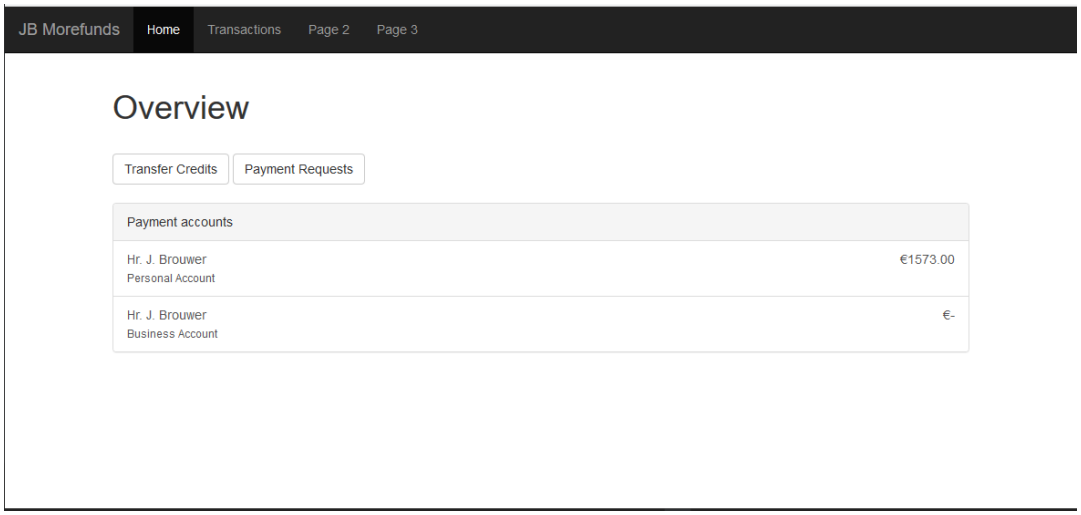

Figure 4.4: The home screen of the website, showing an overview of the bank accounts and the balances.

To enable communication with the front end, the back-end exposes a REST-API endpoint. The front-end uses HTTP-calls to query the REST-API hosted by the application. the REST endpoint currently supports four calls: 'account', 'partner', 'transactions', and 'startTransaction'. An overview of these calls and the corresponding replies can be seen in listing 4.1

4.4. Business Logic

This module contains the 'Business Logic', the code that describes the functionality of the application. It constitutes what a transaction is, and the functions to validate them. In this proof-of-concept, the main feature is the ability to send funds from one client to another. Therefore the business logic is rather straight forward.

One of the guarantees the logic must offer is that money is never created or destroyed, preventing inflation or deflation. This is ensured by the ruling that a sender's new balance is the old balance minus the transaction value and that the receiver's new balance is the old balance plus the transaction value. Second of all, funds owned by a party should only

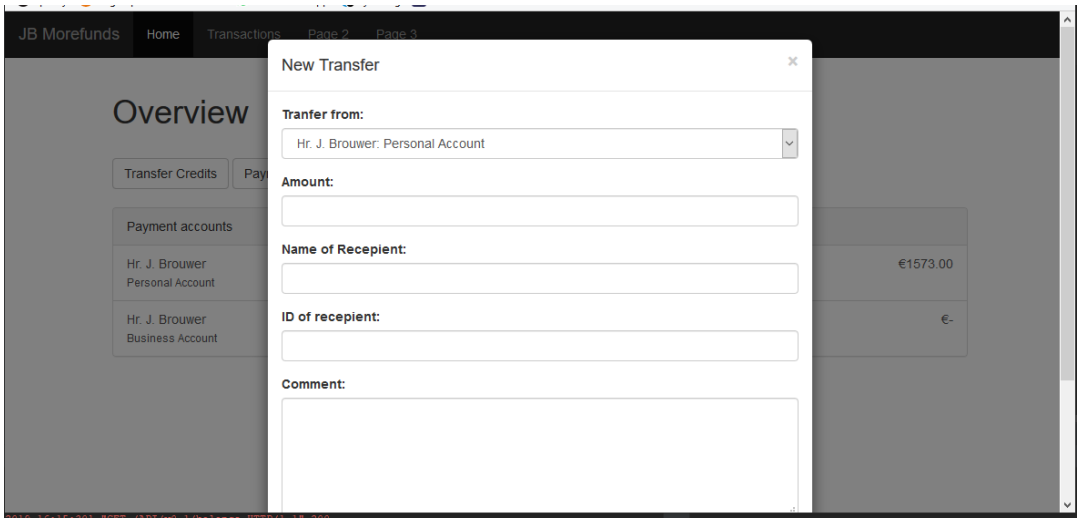

Figure 4.5: The view from which the user can initiate a new transaction.

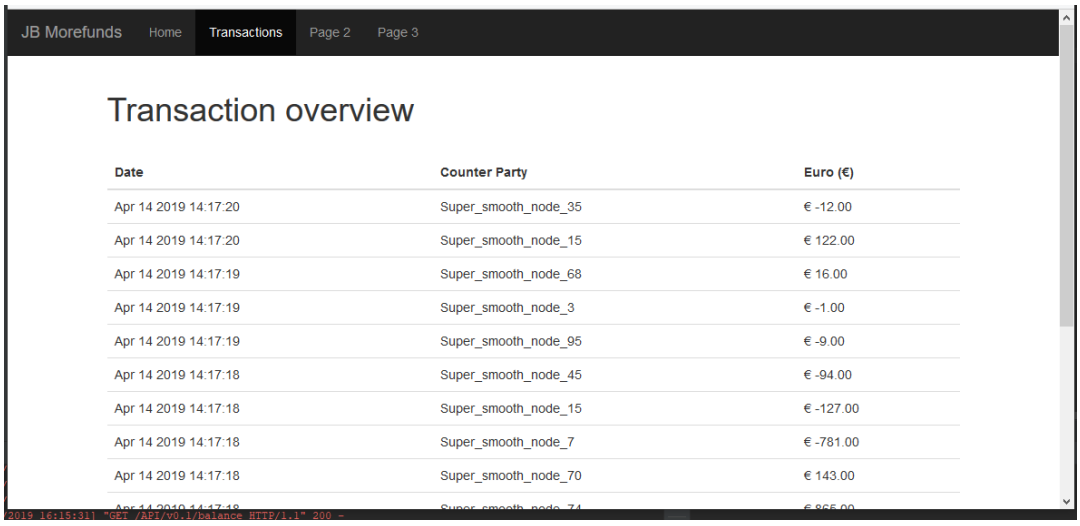

Figure 4.6: An overview showing the most recent transactions.

```
GET '/API/v0.1/account' {}
response {
    Name: Client's name (String),
    Type: type (personal, bussines, etc.) (String)
    Address: encoded in base64 urlsafe (string)
    Balance: Current balance (Int)
}

GET '/API/v0.1/partners'
response {
    Node: [Address_1, ...,Adress_n] encoded in base64 urlsafe (string)
}

POST '/API/v0.1/startTransaction' {
    Recpient: address of counter-party encoded in base64 urlsafe (string)
}

GET '/API/v0.1/transactions?from=block_hash&length=number_of_transactions'
response {
    transactions: [
        (timestamp_1 (String),
        counterparty_1 (String),
        value_1 (String)),
                ...
        (timestamp_n,
        counterparty_n,
        value_n)]
}
```

Listing 4.1: The implemented REST-api calls and there corresponding responses.

be controlled by that party, resulting in only allowing positive transactions since a negative transaction value is essentially withdrawing money from someone else their account. Another requirement is that balances should always be positive, as clients would otherwise be able to act as lenders.

Following all these requirements, a transaction is considered valid if and only if:

- The transaction value is positive.
- The transaction value is smaller than or equal to the balance of the sender.
- The sender's new balance is the old balance minus the transaction value.
- The receiver's new balance is the old balance plus the transaction value.

All these requirements are embedded in a function that takes a transaction, applies these rules, and returns true are false. This function will then be called by the core every time a transaction needs to be validated. This enables developers to create applications on top of this software without having to modify, or even understand the core.

4.5. Trustchain Core

The most widespread implementation of Trustchain is situated in Tribler. During the implementation in Tribler, some design choices were made to tailor Trustchain to the needs of Tribler, but that might not be suited for this proof of concept.

4.5.1. Half-blocks

Tribler uses Trustchain as an accounting mechanism for the provided and consumed bandwidth during file-sharing. As with any other torrent client, multiple peers may be used when downloading a single file. However, this means that a single client might need to create mul-

tiple concurrent Trustchain transactions. When only allowing one transaction at a time, a delay in a single peer can prevent the creation of further blocks. This means that the maximum throughput is limited by the time it takes to conclude a single transaction. When creating blocks there are two major sources of latency:

- Communication induced latency: Latency due to messages/packet delayed or dropped
- Negotiation induced latency: Latency which finds its origin in reaching agreement on the details of the trade.

Instead of minimizing the impact of either of these latencies, Tribler circumvents this problem altogether by using half-blocks.

The half-blocks essentially are trade-proposals which are directly added to the proposer's chain. If the counterparty chooses to accept this trade, he can then create the other half of the block, which will reference the first half. An unresolved trade proposal does not prevent any new trades from being initiated. In Figure 4.7, a graphical representation of two ledgers joint by two half-blocks are shown.

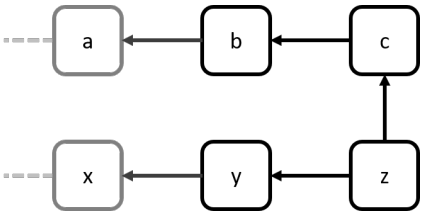

Figure 4.7: Graphical representation of two individual chains linked together through a common transaction.

Since the creation of the half-block does not depend on the counterparty, there is no communication or negotiation overhead, resulting in a high throughput.

Problem of Verification

While this seems to solve the problem of unresponsive or slow counterparties, it only delays the consequences of it. The question now is, how to deal with half-blocks while verifying a chain. A trivial solution is to treat a chain with uncompleted half-blocks as invalid. However, this results in undoing all the benefits of the half-blocks, as one would only append a half-block if the other half is already signed (nobody would risk the change of an invalid chain). Thus, only a new transaction can be started if the previous is successfully completed. We require a way to handle invalid blocks without rendering the chain invalid. When assessing the current state of the chain, the half-blocks should be treated sensibly, even more so if we would want to store the state in every block. There are two options: Include the transaction directly in the current state or wait until the other half is received.

- Directly updating the current state: When including the transaction directly into the current state, the current state becomes worthless: Unless the complete chain is analyzed to determine which portion of transactions are valid, there is no way of knowing the actual current state. (This is the current implementation in Tribler concerning the 'running total'.)

- Update after receiving the corresponding counterpart: When including the transaction in the next block after receiving a signed counterpart, transactions become difficult to verify, as the current state seems to update with a randomly seeming offset. Even when referencing the block that supposedly got fulfilled, there is no way of knowing if that block ever existed and was part of the chain unless the verifier walks the chain until he finds it (or reaches the genesis block if it does not exist).

In both cases, the benefit of being able to determine the state without verifying the complete chain is lost.

Life Without Half-blocks

The issues associated with the verification can be circumvented by not using half-blocks in the first place. A visual representation of two ledgers being linked by a jointly created block is shown in Figure 4.8. Doing so might slow down the block creation but improves block verification as transactions can be checked in a single lookup, and uncertainty is removed: Either the block is invalid and always will be, or it is valid and always will be.

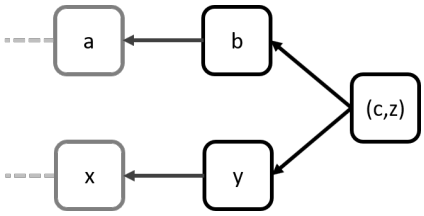

Figure 4.8: Graphical representation of a jointly created block.

If no mutations to the chain are required while negotiating a trade, multiple trades can be negotiated in parallel, increasing the throughput of the network. An argument against parallelization is that by preparing multiple transactions concurrently, one is essentially creating temporary forks on their own timeline. Which opens up a whole variety of potential problems; an example of this is given in the Figure below:

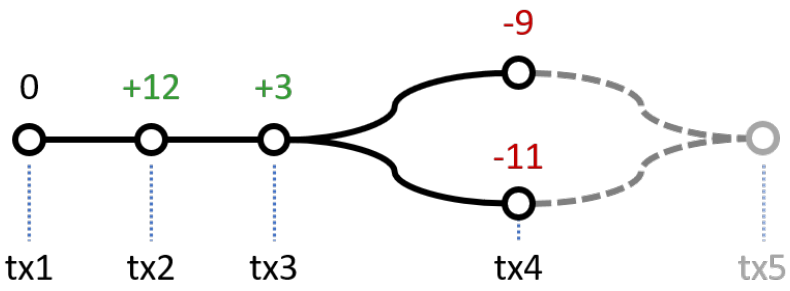

Figure 4.9: Schematic overview of a transaction-timeline with a fork due to concurrency.

In Figure 4.9, we have transactions of units seen from a single node's perspective. A positive number means a transaction in which the receive units, where negative values are transactions in which they spend units. This example assumes that one is not allowed to have a negative unit balance.

TX1 to TX3 pose no threat, but problems start to arise at TX4. At the initiation of TX4, both parties might not know of the other party's transaction in progress, and both seem valid transactions on their own. The total sum of tokens becomes negative, which violates the requirements of always having a positive balance (i.e., a double-spending attack).

While some solutions might be possible, it is anything but trivial (Checking the chain after creating but before committing results in a race condition, and rendering one of them invalid is tricky; if transactions can be revoked later no node can ever trust another node on their balance)

The problem mentioned above lies mainly in the fact that every time concurrent transactions occur, a temporary fork is created, but parties are unaware of the other forks. If concurrency

is required (for example, due to slow negotiations), explicitly mentioning the fork will solve the problem. When creating an explicit fork, both parties sign a block indicating they are engaging in a transaction, this block also includes the maximum transaction volume associated with the fork. By doing so, it is guaranteed that the value associated with the current fork cannot be spent in another branch of the chain. When the transaction is completed, a transaction-block is created, and the fork merges back with the rest of the chain.

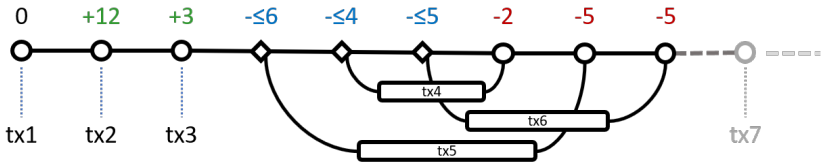

Figure 4.10: Schematic overview of a transaction-timeline with explicit forking with three concurrent transactions.

The example in Figure 4.10 uses the same concept as Figure 3; however, now it creates a block to indicate a fork in the timeline explicitly. Due to this, tx4 and tx6 can be initiated while tx5 is still in progress. When initiating a new transaction, it becomes directly clear what the current state of the chain is.

The number of concurrent transactions can be limited by not allowing the oldest fork to be older than X blocks. This significantly increases the verifiability of a chain since the maximum number of blocks that need to be checked before ensuring there are no open forks $= X \times 2$.

For this proof-of-concept, it was chosen to not allow any concurrent transaction at all, to focus on the performance of fair witness selection protocol.

4.5.2. Serialization

The blocks and the transactional data within those blocks should be serializable to enable the storage and transport of this data. The Tribler implementation uses a table in an SQLite database to store this data. The scheme used to store this data is shown in Figure 4.11. This choice is mostly motivated by legacy code and ease of prototyping, however, it suffers in flexibility and efficiency.

Adding, removing, or changing the type of a field would require changing the database schema and converting every block from the old to the new form. This would not only require much computational power since each peer can have thousands of blocks, but it also breaks backward compatibility, which can have disastrous consequences because of Tribler's distributed nature and no way of enforcing updates.

The method of storing the data also suffers from inefficiencies. The fields 'public key', 'link public key', 'previous hash', 'signature', 'block hash' for example are all stored as text in a hexadecimal representation. A single byte is displayed using two alphanumerical characters meaning that every byte data is stored using 2 bytes, resulting in a 50% overhead.

To offer some flexibility, the actual transaction data is stored in plain-text using JSON notation. While this does offer the option to change the format of a transaction without breaking backward compatibility, it comes at the cost of size. The transaction in Figure 4.2, for example, uses 86 bytes to store 32 bytes worth of data.

Since this proof-of-concept does not require to be compatible with the current Tribler implementation, the serialization and storage method can be reconsidered. The methods that were considered are JSON, Protocol Buffer, custom binary format, and Python's Pickle.

Field	Type
type	Text
tx	Text
public key	Text
sequence number	Text
link public key	Text
link sequence number	Integer
previous hash	Text
signature	Text
block time stamp	Integer
insert time	Integer
block hash	Text

Figure 4.11: Database schema used in the current Implementation in Tribler.

```
{
    "down": 29493460,
    "total_down": 1811019874262,
    "up": 0,
    "total_up": 34505368528
}
```

Listing 4.2: A transaction taken from the current Tribler implementation.

JSON JSON is an open-standard file format that stores data in a human-readable format. The notation is derived from JavaScript, hence the name JavaScript Object Notation. It was designed to be used for Server-to-browser communication and does so using objects described in key-value pairs. The benefit from this is that no predetermined format is required, as any object can be easily added to the messages, offering both ease of use and flexibility. Since JSON is widely and used and uses plain-text, most mainstream programming languages either have built-in support or well-written libraries, JSON is very portable across both languages and platforms. A downside of JSON is that everything is stored using plain-text. For storing strings, this poses no problem. However, numbers need to be converted to their base-10 representation even for floating-point numbers. In the worst case, a double-precision float (8 bytes) would contain 767 significant numbers, thus requiring 767 bytes. Since trustchain also has to store signatures and hashes, which are all bytes, they either have to be represented in hexadecimal notation (2 bytes for every byte) or base 64 (4 bytes for every 3 bytes).

Protocol Buffers Protocol Buffers (Protobuf in short) is designed by Google to be performant yet straightforward, and be both platform and language agnostic. Protobuf is used by first defining a schema similar to defining structs in C in a '.proto' file. This proto-file is compiled to generate code for the desired programming language that is used to create, serialize, and de-serialize messages. Google provides generators for C++, Java, Python, Go, Ruby, Objective-C, C#, and JavaScript. Messages can be serialized in a specific language and can be de-serialized in any other language, offering excellent portability. While Google claims that Protobuf messages are backward compatible, they generally mean that an instance can send messages using a newer format without breaking instances that are still running code generated using an older schema. After serializing, the data is stored in a binary format, resulting in small messages after serialization.

Custom format An easy and efficient solution might be to simply concatenate the bytes that make up a block in a predetermined format. This would be efficient both time- and space-wise as it has no overhead at all. When defining a custom format, portability is trivial. However, it does require writing a parser for every language that needs to be supported. When needing to support dynamic length data types such as arrays, a custom format becomes a bit more complicated. Flexibility is also very limited, as changing the format would require software updates for all instances, and all blocks would have to be converted to the new format.

Python's Pickle Python natively offers a method of serialization of objects for storage and transfer called Pickling. All native data types or composition of data types can be 'Pickled'

using the built-in library and a single function call. While Pickling offers simplicity, flexibility, and space efficiency, it can only be consumed by Python instances offering essentially no portability at all.

	Simplicity	Flexibility	Space Efficiency	Portability
Json	+	+	-	+
Protobuf	+/-	+	+	+
Custom format	+	-	+	+/-
Python's pickle	+	+	+	-

Table 4.1: Overview of considered serialization methods and their properties.

Looking at Table 4.1, both Python's Pickle and Protobuf seem good contenders. The pickling method outshines Protobuf when it comes to simplicity because it does not require additional tools or a different language to define schemas. Nevertheless, Protobuf's platform and language agnostic properties open the door to implementing smartphone, web, or even embedded-hardware clients. Therefore Protobuf was considered the best choice for serialization.

4.5.3. Block format

The block and transactions are split up into two separate Protobuf messages. The reason for this is that a transaction needs to be signed before it can be sent to the witnesses. Protobuf uses the concept of 'varints,' which gives the possibility to store a 64-bit integer with fewer bytes if the value could be represented with fewer bits. The number 1 would usually be encoded as 0x01, but it may also be encoded as 0x8100 or 0x818000 or 0x81808080808000, as the specifications state that the shortest 'should' be used instead of 'must' be used. This results in a possibility that different platforms or languages may not result in the same bitwise perfect copy of serialized messaged. While the difference does not prevent the message from being interpreted on any other platform, it does create a problem with regarding the signatures, as a single bit difference renders a signature invalid. However unlikely, this risk was not taken.

The used format for a block is shown in listing 4.3.

```
message Block {
    bytes transaction = 1;
    repeated bytes signatures = 6;
    repeated Approval witnesses = 2;
}

message Transaction {
    uint64 timestamp = 1 ;
    int64 value = 2;
    repeated bytes ID = 3;
    repeated bytes previousHash = 4;
    repeated uint64 sequence_no = 5;
    repeated uint64 balance = 6;
}
```

Listing 4.3: excerpt from the '.protofile' defining the transaction and block format.

5

Experiments and Evaluation

The main goal of this work, as described in Section 2.4, is to create a truly scalable distributed ledger that does not rely on global consensus. While the correctness and security have been covered in Section 3.3.2, the scalability has yet to be verified. To do so, several experiments have been carried out to see how the system scales. Besides scalability, also the performance of the network under adversarial influences is evaluated.

5.1. Experiments

5.1.1. Setup of the Experiment

In order to determine the scalability of the proposed solution, the DAS-5 was used, a compute cluster consisting of 198 compute nodes spread out over six physical locations [39]. Each compute node is equipped with dual 8-core 2.4 GHz (Intel Haswell E5-2630-v3), 64GB of ram, 128Tb of storage, and was running CentOS Linux.

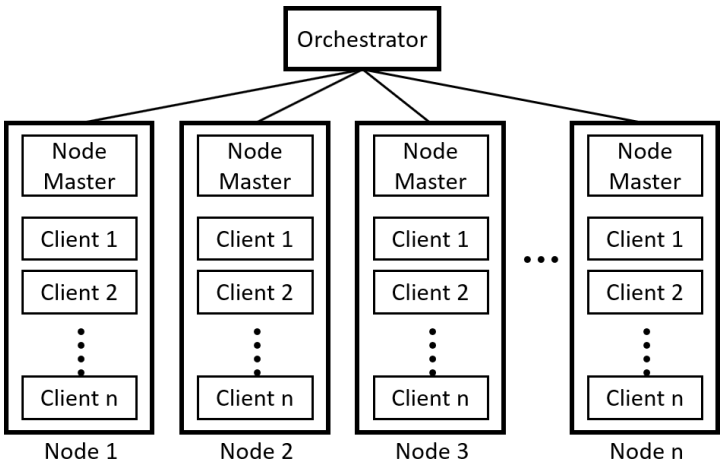

Figure 5.1: The architecture of the testbed depicting multiple compute nodes, each running a Node Master jointly controlled by the orchestrator.

To control the potentially thousands of nodes, a so-called 'Orchestrator' was written. This orchestrator is the command center, instructing the nodes when to start or stop the emulation using which parameters. Instead of launching thousands of nodes on tens of compute nodes

directly, the orchestrator starts an additional program on each compute-node that launches the required nodes on the orchestrator's behalf. This node master also ensures that all the created processes are terminated to prevent orphaned processes in case of a crashed node. An overview of this architecture is shown in Figure 5.1.

One of the parameters passed to the nodes is the IP address and port number, as soon as the node is ready, it connects to the orchestrator and announces its presence. As soon as all nodes called in with the orchestrator, it signals the nodes which experiment to run and for how long.

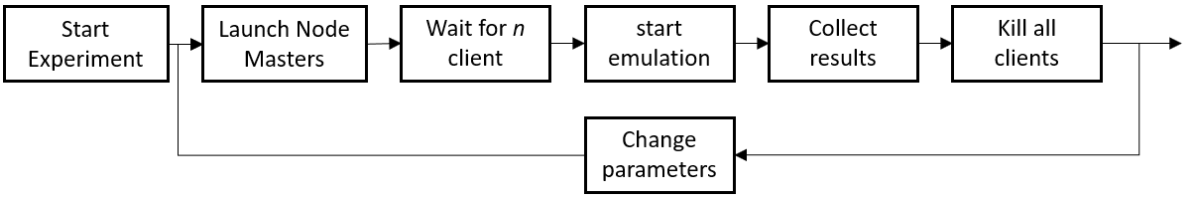

Figure 5.2: The general chain of actions used when running the emulations.

After a node finishes its experiment, it reports its results to the orchestrator. If a (configurable) threshold of reports is reached, a cool-down period starts, to give slower nodes time to catch up before killing the process, while not waiting indefinitely in case a node would crash. An overview of the emulation process can be seen in Figure 5.2.

5.1.2. Scalability

As the block signing is a one-shot function and the number of required witnesses is largely invariant to the network size, the system should be highly scalable in terms of throughput and latency. Here throughput refers to the network's total number of transactions per second, whereas latency refers to the total time it takes from initiating a transaction up to a fully validated and verified transaction.

The implementation used to perform these tests is an adaption of the implementation, as described in Chapter 4. The web-interface and REST-API endpoint were disabled to reduce the computational overhead, as these are not needed for the test. One change to the protocol is that, during the establishing of the transaction, the receiving party may change the direction of the transfer. Clients will always choose the direction in such a way that money flows from clients with a high balance to clients with a low balance. This change is without loss of generality and is done solely to ensure that clients can trade for the entire duration of the experiment.

All experiments are run several times to measure consistency as well as to minimize the impact of hardware or operating systems flukes.

Throughput

To asses the network's total throughput, every client is configured to initiate a transaction with a randomly selected partner spontaneously. After every transaction, the client will sleep for X seconds, where X is drawn uniformly from $(0,1)$ resulting in an average waiting time of 500 ms between transactions. The same order of magnitude as the human reaction time.

The experiment will run on twenty compute nodes. Every compute node would then be ordered to host five clients resulting in a total of 100 clients. After every successful run, each compute node would be ordered to increase the client count by five, resulting in steps of hundred clients.

Contrary to the expected linearly growing throughput, the result in Figure 5.3 shows a saturation. Due to this non-linear behavior, consecutive runs were canceled, to first investigate this unexpected behavior. When extrapolating the linearity of the first 30 measurements (10 - 700 nodes), the deviation from linear growth becomes even more apparent.

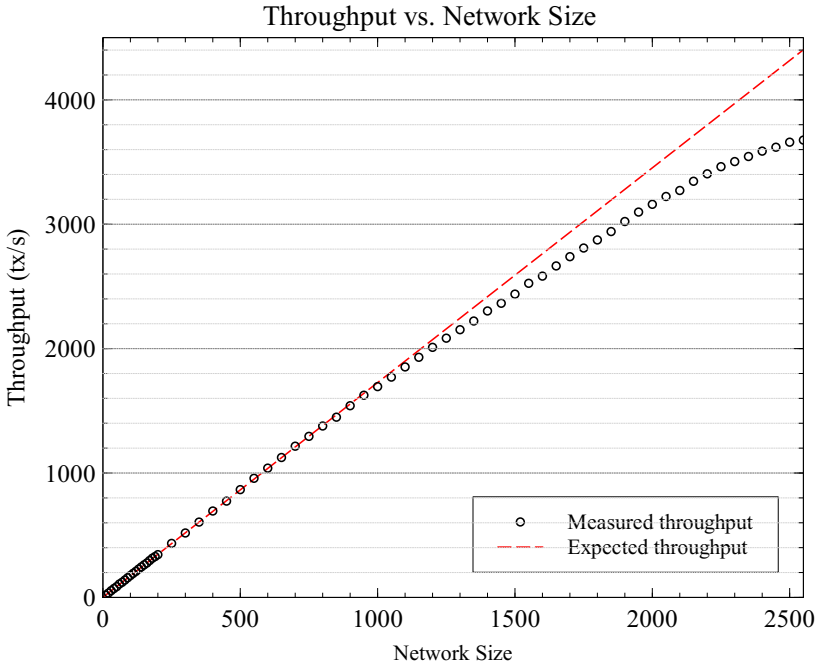

Figure 5.3: Throughput of an early prototype containing a highly un-optimized witness selection implementation. Due to the clearly non-linear behavior, consecutive runs were postponed.

As the algorithm is designed to have a constant run-time with regard to the network size, the non-linear scaling must arise from either the implementation or from the hardware. Since real programmers never doubt their software, the hardware was observed during the emulation. None of the resources were used enough to have a significant impact on the throughput; The CPU, Memory, and network utilization were well below 50% of their respective limits.

Therefore it is more likely that the error lies within the implementation. After thoroughly going through the source code, an unoptimized implementation of the equation for selecting the witnesses was found. Due to Python's inability to directly compare byte-arrays, a custom, but naive method for selecting the witness was implemented. This naive method resulted in a function with a run-time complexity of $O(n)$. Early on in the process, this function was implemented and tested on correctness. However, the time complexity of this function was not evaluated.

Even though the total time spent in the function was relatively small, the function is run for each witness the initiator requires, and then again by every selected witness to verify before replying to a witness request. The multiple sequential calls combined with a growing number of potential witnesses to select from may lead to significant performance loss.

After replacing the witness selection with a binary search, the run-time complexity was reduced to $O(\log(n))$, and the experiments were rerun. The results are shown in Figure 5.4. The results do not only show the average throughput but also the consistency over multiple runs, in the form of error bars. This time the results seem significantly more linear than the previous iteration.

A straight line is fitted through the data points, and the root-mean-square error was used to determine the linearity. For the corrected version, it yields an RMSE of 0.68%. When doing

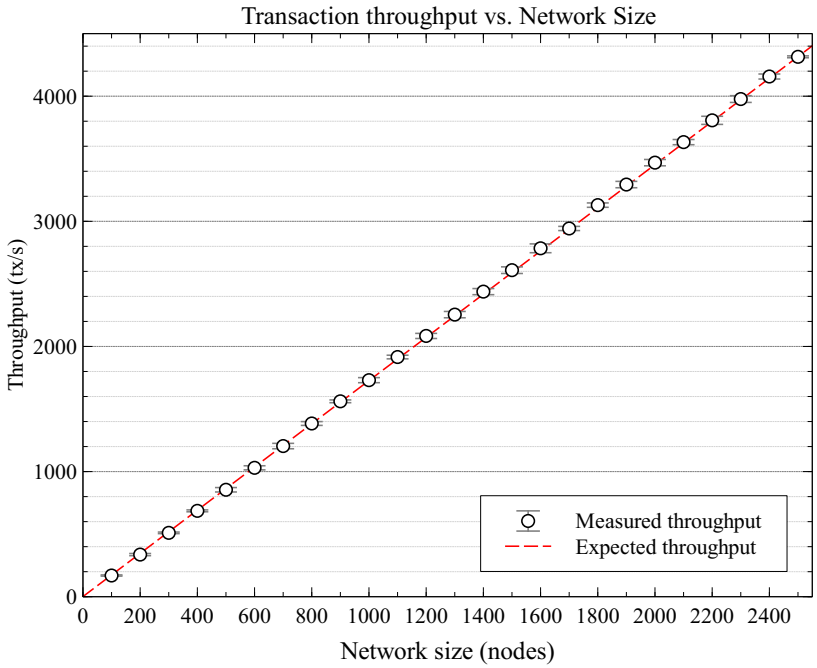

Figure 5.4: The network's total throughput as function of the network size. The width of the error bars was set to be extra wide, as otherwise, the markers would render them invisible them.

the same for the first iteration and using the extrapolated trend line plotted in Figure 5.3 gives an RMSE 7.12%. Furthermore, the first iteration would yield increasingly larger error for increasing network sizes, as the divergence from the expected throughput would only grow.

While the randomized test gives a more realistic workload, it does not put the machines or the algorithm to the fullest. It was decided to perform an additional experiment similar to the first one, to determine the maximum throughput for a given machine. However, instead of waiting for a random amount of time and then select a partner at random, every party selects a fixed partner and only interacts with them.

the partner selection is done relatively even: $if\ n = odd \implies p = n - 1, else\ p = n + 1$. Where n is the index of the client in the sorted client list, and p is the index of the selected partner in the same list. Further, only even-numbered nodes are allowed to initiate a transaction to prevent partners initiating transactions at the same time.

Just as before the experiments are run on the same twenty dual-8 core compute nodes. Since there will be no idle time between transactions, the CPU utilization is expected to increase significantly faster. Therefore the number of clients will be increased in steps of 40. The result of this experiment is displayed in Figure 5.5. It can be seen that in the first five increments, the throughput increases by roughly 1000 tx/s per 40 new clients. After the sixth increment, at a network size of 280 clients, the throughput increases by less than 400 tx/s up to 6394 tx/s. Adding another 40 clients even reduces the performance to 6357 tx/s.

Monitoring the resource utilization learns that, at 100%, the CPU was the bottleneck. This does not mean the system cannot scale beyond 6394 tx/s; it just means that more, or more powerful compute nodes are required when a higher throughput is desired.

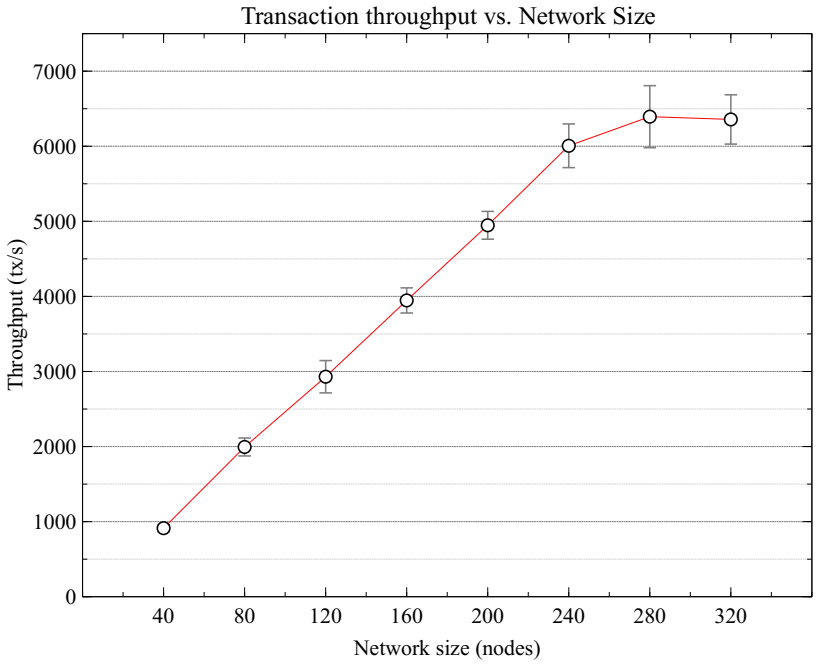

Figure 5.5: The network's total throughput as function of the network size, in case of the fixed partner simulation. The markers denote the average throughput per network size, whereas the error bars indicate the standard deviation.

Latency

Throughput plays an important role in the performance of a distributed ledger, but it does not tell the complete story. An example is the visa network, which claims a capacity of 65.000 tx/s, yet confirmation time can be several days [40]. Resulting in the funds not being accessible by either of the parties while in transit.

The same setup as for the previous experiments was used to evaluate the performance of the network. The only modification is that a single node is chosen to record the latency of each transaction they initiate. This was implemented using the system clock to record the timestamp of sending the first message, and again after the transaction was validated and signed by both the counter-party and witnesses. This way, the total transaction time, from initiation to finality, could be easily and efficiently measured.

The resulting transaction times are plotted in Figure 5.6. The box shows the interquartile range (IRQ), whereas the whiskers show $1.5 \times IRQ$. The circles show the average, and the crosses display outliers. Here it can be seen that all transactions complete within about 15 milliseconds. A key take away from this graph is the invariance of the transaction time with regards to the network size. No clear explanation was found for the few outliers seen in the measurements, but there does not seem to be a clear connection between them and the network size.

5.1.3. Small-Scale Experiments

For comparative evaluation, the same experiments were executed again, but now with similar parameters as Blockbench [37], an evaluation framework for distributed ledger performance. By choosing similar parameters, it will easier to make meaningful comparisons with the results presented in the paper.

In Blockbench, an experiment was done where each compute node acts as both a client and

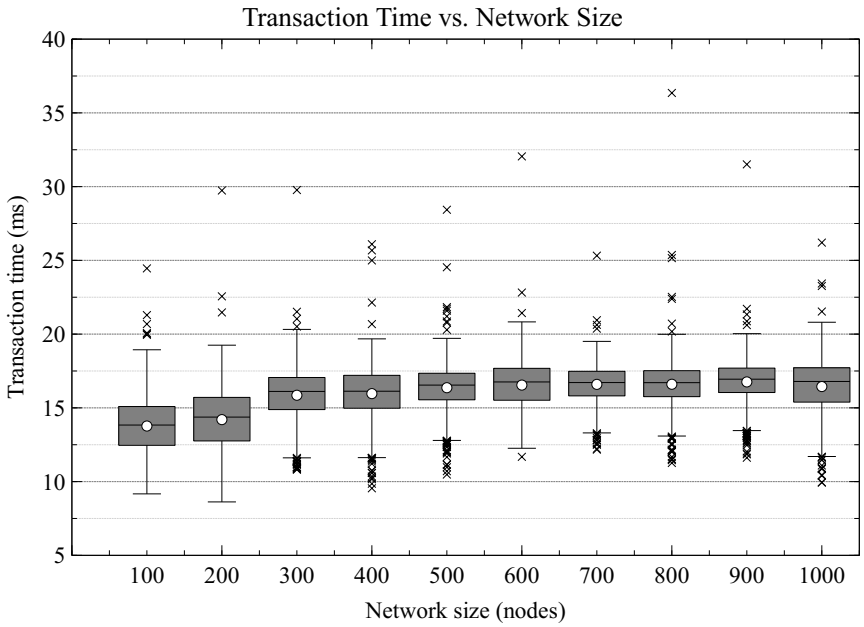

Figure 5.6: The transaction time in milliseconds, as a function of the network size. Here the transaction time constitutes the total time from initiating a transaction until having a fully signed and validated block. The box shows the interquartile range, the whiskers show 1.5× the interquartile range. Statistical outliers are denoted by the crosses

a validator, similar to this implementation. The chosen number of clients/validators ranges from 4 to 32 in increments of 4. To mimic this behavior, the same method for selecting partners as earlier were used, but now with only two clients per compute node. The number of compute nodes was increased in steps of two up to 16 nodes, resulting in a range from 4 to 32 clients. The results of these experiments are shown in Figure 5.7 and Figure 5.8.

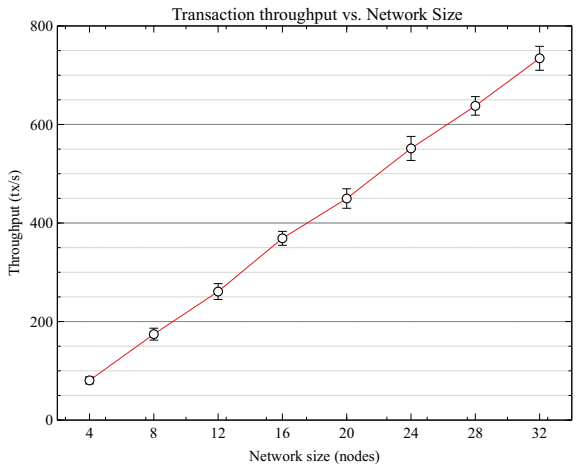

Figure 5.7: The total network's throughput as a function of the network size when run at a smaller scale.

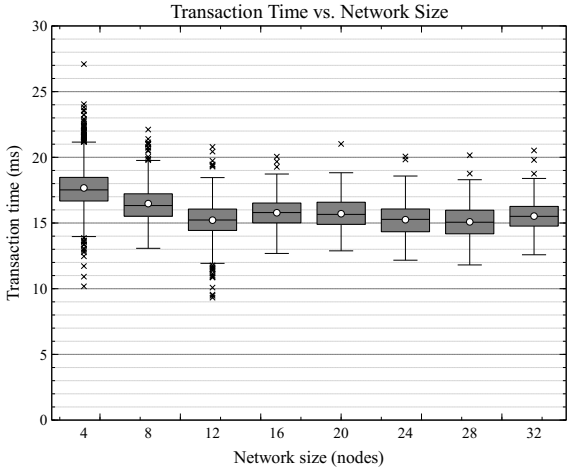

Figure 5.8: The transaction time as a function of the network size when run at a smaller scale.

5.1.4. Performance Under Adversarial Influence

So far, all the experiments were under the assumption that no malicious actors were involved. And while the security of the algorithm has already been proven in Section 3.3.2, it does not tell us anything about the performance under adversarial influence.

In this experiment, we focus on a denial-of-service attack on the network. This could be done by either directly controlling the nodes and simply refusing to witness transactions, or as disrupting the witness process indirectly by performing a denial-of-service attack on potential witnesses.

To emulate this behavior, the same experimental set-up as the latency experiments is used. An increasing portion of the network becomes malicious and refuses to participate in the witnessing process. A threshold variable is passed to the node to determine if it is malicious or not. At startup, it then checks if its ID is greater than the threshold to figure out what kind of behavior it should exhibit. Instead of directly replying with a NACK to the request, a malicious witness accepts the incoming request and then lets the connection time out to frustrate the process as much as possible.

As the maximum throughput is the reciprocal of the sum of the transaction time and sleeping time, it was chosen not to perform any throughput experiment. The resulting throughput would, in this case, be an indirect measurement of the latency, but would be less valuable information as it also includes the random waiting time.

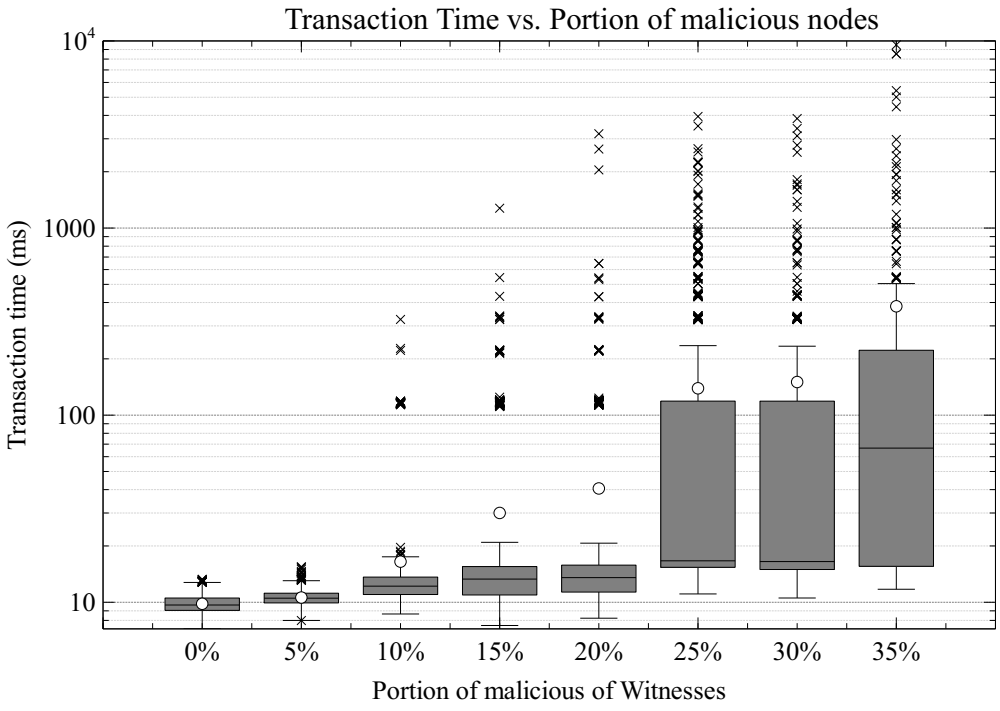

Figure 5.9: The transaction time in milliseconds as a function of the increasing portion of malicious actors in the network, shown in a box plot. The box shows the interquartile range, and the whiskers show 1.5× the interquartile range. The crosses denote statistical outliers

The results of this experiment can be seen in Figure 5.9. The boxes show the interquartile range of the latencies, whereas the whiskers show $1.5 \times IRQ$. The circles denote the average latency, and the crosses depict the outliers. While it may seem that the system still functions, albeit poorly, beyond the theoretical upper bound of $\frac{1}{3}$ malicious nodes, this is not the case. Where 100% of the transaction succeeded up to 30% malicious nodes, less than 10% of the transaction succeeded in the case of 35% malicious nodes.

5.2. Evaluation

The design goal of this work was to create a secure and scalable ledger that does not rely on global consensus. The security aspect is already addressed in chapter 3, so in this section we will evaluate the performance and scalability of this solution. Scalability in this work was defined as:

1. *Scalable in the number of validator nodes*: There should not be a limit on the number of validators that are supported by the system, neither should the throughput decrease when more validators are added.
2. *Scalable in the number clients*: There should not be a limit on the number of clients; neither should the throughput decrease when more clients are added.
3. *Scalable in transaction throughput*: There should not be a theoretical upper bound due to algorithmic constraints.

When looking at Figure 5.4, it is fair to say that the network sees no degradation in throughput as the network grows, and instead steadily keeps increasing. In Figure 5.6 the latency also displays no degradation as the network grows. The RMSE (with respect to a straight line) is 0.68%. The average deviation of multiple runs for each network size is, on average, 0.93%. The non-linearity of 0.68% is well within the margin of error, and it is, therefore, safe to say that the system scales linearly (at least up to 2500 nodes).

The same scalability can be observed when looking at the latency in Figure 5.6. Except for the network sizes of 100 and 200 nodes, the latency is consistently around 16 ms regardless of the network size. A possible reason is that the small increase in transaction time can be attributed to the witness selection implementation, which runs in $O(log(n))$ run time. One could argue that the average latency does indeed show a logarithmic curve. However, due to the minor impact on overall performance, this was not further researched, and can therefore not be said with certainty.

In Figure 5.9, it is shown that even under adversarial influence, the system is still fully functional. When looking at 20% of a malicious portion, the average latency remains below 40 ms. The 50th percentile of the transactions is even finalized within less than 20 ms. For the cases of 25% and 30% malicious portion, the average transaction time lies around 105 ms.

Looking back at the design goals, it is safe to say that scalability in validator nodes, clients, and throughput are all achieved. With regard to security, it is shown that the network keeps making progress even up to the theoretical bound of $\frac{1}{3}$ malicious actors.

5.2.1. Comparison

Due to the unique architecture of this solution, an apples to apples comparison with existing work is not straight forward. It is difficult to create a level playing field to compare performance with existing work since commonly used existing solutions all rely on global consensus, and none of them utilize pair-wise ledgers.

However, the claim of this work is not to create the highest throughput or lowest latency ledger, but to design a scalable and secure ledger. So instead of directly comparing the performance of different solutions, we will look at how the performance of each solution develops as the network scales.

The results achieved in this work will directly be compared to the results found in literature. While one could argue this would not be a fair comparison due to different hardware configurations, it is clear that the computational power of the hardware the run-time complexity,

and therefore the asymptotic upper bound will not magically decrease.

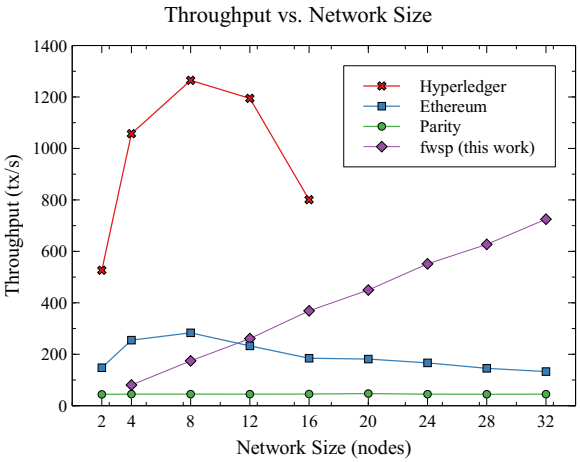

Figure 5.10: Total throughput as function of network size. The number of validator nodes equal the number of clients. Figure adapted from [37].

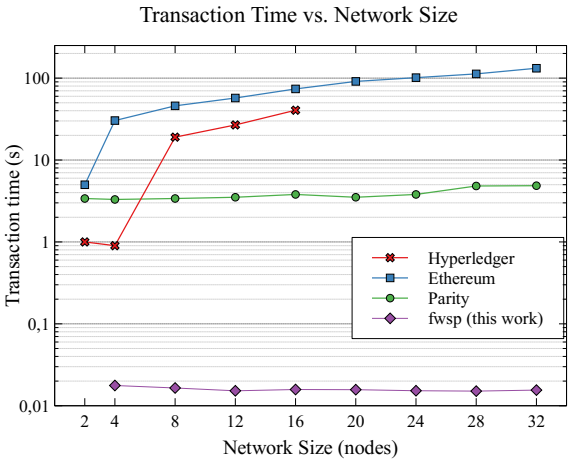

Figure 5.11: Transaction time as function of network size. The number of validator nodes equal the number of client. Figure adapted from [37].

In [37], the three most mature platforms (Hyperledger Fabric, Ethereum, and Parity) were evaluated on throughput, latency, and scalability. The results acquired by the team are displayed in Figure 5.10 along with the results from this work. As shown in the results, it is clear that the evaluated existing solutions either decrease as the size reaches beyond a specific size, or remain stable regardless of the network size. The solution proposed in this work clearly shows a linear growth with the network size, resulting in superior scalability.

Figure 5.8 shows the latency of the proposed solution to major commercially available platforms. Here it can be seen that the latency of this work is multiple orders of magnitude lower than the rest, while also remaining stable with respect to the network size. Even under the strongest adversarial influence this work can tolerate, it still outperforms all other solutions by order of magnitude with an average latency of 105 ms.

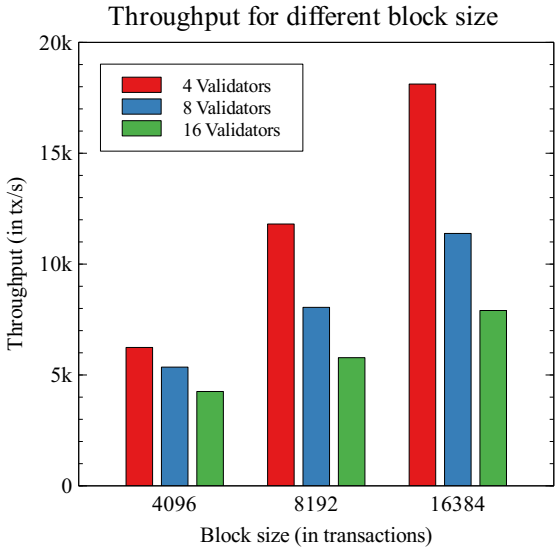

Figure 5.12: Tendermint's throughput for different block sizes for a different number of validators. (Figure adapted from [38])

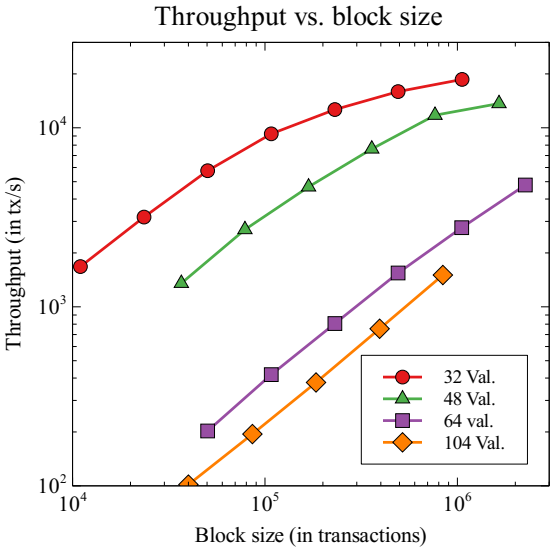

Figure 5.13: Throughput as a function of block size for a different number of validators. (Figure adapted from [13])

Most solutions focus on winning the throughput arms race, so little evaluation as done with

regards to scalability. In [38], experiments were done to determine the impact of different block sizes on Tendermint's throughput, for a different number of validators. Data points from multiple runs with different numbers of validators were taken and plotted in Figure 5.12. The same was attempted for HoneybadgerBFT. However, no multiple data points existed for the same block size. Instead of compromising accuracy by interpolating the data, it was chosen to plot the original data as a different series (Figure 5.13).

For both Tendermint and HoneybadgerBFT, it is clear that increasing the validator set has a significant negative impact on the throughput of the solutions where this work displays a linear growth with respect to an increasing network size.

Despite running on an older generation CPU and only half of the RAM, Hyperledger (while not scaling very well) still yields significantly higher throughput. It only requires eight validator nodes to reach close to 1300 tx/s, whereas our solution seems to require somewhere between 40 and 80 nodes.

This is due to architectural differences between the solutions. By design, a client in this work is only allowed to start a new transaction when the previous is validated. In contrast, a client in Hyperledger Fabric is allowed to submit transactions as fast as they can or even submit them in batches. For the experiments to be as similar to [37] as possible, an equal number of clients and servers was chosen. This results in the throughput being limited by the number of clients, instead of the number of validators. This same reasoning holds for a Tendermint and HoneybagerBFT.

A more interesting quantitative comparison is to the experiments by the authors of Algorand. This is due to the fact that Algorand is designed to be extremely scalable with regards to both clients and validators (similar to this work, both roles are collapsed to one node). In [33], it is shown that the network easily scales up to 500.000 nodes, while retaining a constant latency.

When assuming the number of byzantine nodes to be $1/5^{th}$ of the complete network, just as Algorand did for their experiments, our solution requires a witness set size of only 38 witnesses to guarantee a probability of $< 5 \times 10^{-9}$. Algorand required 2000 nodes to give the same guarantees.

In their experiments was no notion of a transaction, but they expressed their throughput in the form of megabytes per hour. For a block size of 10MB (the largest tested) and 500.000 users run on 1.000 machines, a throughput of 750 MiB/h was achieved. This work achieved an average throughput of 15.8 GiB/h With only 20 machines running 280 nodes.

5.2.2. Summary

It is shown that the performance of Hyperledger, Tendermint, and HoneybadgerBFT is significantly reduced when increasing the number of validators, meaning they are not scalable in terms of validators. Furthermore, it is shown in [37] that an increase in transactions results in an increase in latency, meaning it also does not scale in throughput.

Ethereum (and any other Nakamoto based ledger) scales both in the number of clients as well as validators. However, by design, the amount of blocks (and therefore the throughput) is limited as this is what guarantees the security of the ledger.

While Algorand is the most scalable BFT solution in the number of validators by fixing the number of participants in the agreement phase, there is a lower-bound on the number of nodes (without compromising security). As the communication complexity of the agreement protocol is dominated by the number of participants, this lower-bound on the number of nodes implies an upper-bound on the performance. Therefore Algorand has an algorithmic bound on the throughput, and therefore indirectly on the latency (If transactions are received

faster than they are processed, they have to wait until the next round).

The solution presented in this work is shown to scale both in validators and clients. Since witnessing a transaction is a one-shot non-blocking operation and two unrelated transactions do not interfere with each other (even if it involves the same witnesses), there is no algorithmic upper bound either.

	Hyperledger Fabric	Tendermint	Ethereum	Honeybadger	Algorand	**FWSP (this work)**
Scales in validators			✓		✓	✓
Scales in clients	✓	✓	✓	✓	✓	✓
Scales in throughput	✓	✓		✓		✓

Table 5.1: Scalability of different solutions compared to this work.

6

Conclusion and Future Work

6.1. Conclusion

A well designed distributed ledger has the potential to let multiple different parties work together efficiently, without necessarily trusting each other. Until now, most ledgers were limited in the number of nodes that are allowed to validate the transactions, leading to asymmetric power distribution among the users, or were limited in the total number of transactions per time frame, limiting the usability of the said ledger. This led to the question "How to achieve a truly scalable secure distributed ledger without global consensus?".

This thesis proposed an algorithm to create a secure distributed ledger without global consensus, and that only requires partial knowledge and partial ordering of transactions. By doing so, a significant bottleneck in scalability was removed. The algorithm presented has an assumed communication complexity of $O(log^{\star}(n))$ with respect to the network. Effectively removing any limits on the number of validators and clients, such that even in large networks, every node can have an equal say in voting power, paving the way for a fair global scale distributed ledger.

It is shown how the algorithm can be deployed against common attacks against ledgers, such as double spending, forking, and block withholding. Equations to calculate the upper-bound of the likelihood that such an attack succeeds are given. It is shown that 128-bit and 256-bit equivalent security can be achieved with a limited set of witnesses and a small number of additional blocks that need verification.

The algorithm does assume that less than $\frac{1}{3}$ of the network is malicious, and is, therefore, only applicable to permissioned networks, or networks that have sufficient measures in place to prevent a single party controlling multiple nodes (e.g., strong identities).

To verify these rather bold statements, a proof-of-concept, in the form of self-sovereign banking architecture, was implemented and subjected to a suite of experiments. The results were compared to commercially available, state of the art distributed ledgers, and it was found to significantly outperform them in terms of scalability, throughput, and latency. The throughput was tested up to 2500 nodes, and was shown to be linearly; a doubling in nodes doubles the capacity of the network. Where Algorand processes 750 MiB/h but requires a committee of 2000 nodes to guarantee security, this work only required 38 witnesses to guarantee similar security while having a throughput of almost 16 Gib/h.

The three design criteria for this work were security, scalability, and no global consensus. When looking at the design and the results, it can be concluded that the work presented in

this thesis meets all of three. Hopefully, this work will help create a new generation of fair, truly scalable, and secure ledgers that can connect economies and societies to benefit all of us.

6.2. Future Work

6.2.1. Value-Privacy

A valuable addition to the system could be value-privacy. With value-privacy, the actual value of the transaction is hidden using some cryptography, but it is still possible to determine the involved parties. One, rather elegant way to do this is by using 'Confidential Transactions' as proposed by Greg Maxwell[41].

Maxwell leverages the similarity between elliptic-curve digital signatures algorithm (ECDSA) and Pedersen commits based on elliptic curves. It capitalizes on the fact that an EC-Pedersen commit on the value zero turns out to be a valid public key in the elliptic curve digital signatures algorithm scheme. Due to the additive homomorphic properties, two commitments can subtracted, and the sum of the blinding factors will be the private key for the corresponding public key if and only if the sum of the values is zero. Using the (private) blinding factors as the secret input for an eliptic-curve Diffie-Hellman key exchange, both parties can then find the joint private key and sign the transaction with it.

Since a valid signature can only be created if the sum is zero (i.e., no assets were created or destroyed) and both parties were involved, a single signature acts as a proof of validity and authorization of the transaction.

6.2.2. Sender/Receiver-privacy

Sender/receiver-privacy protects the identity of the sender, receiver, or both, dependent on the chosen implementation. When having value-privacy, the direction of the transaction is already hidden (unless explicitly mentioned), meaning that there is somewhat a sense of sender/receiver-privacy, as it is not clear in what direction the funds flowed. Thus for a 3rd-party it is impossible to determine the sender and the receiver with a greater than 0.5 probability.

This probability can be decreased by either combining multiple transactions, adding zero-value transactions, or both. If the transactions support value-privacy, a transaction with ten entries might have one to ten actual transactions (or even zero, to frustrate potential adversaries). The probabilities of guessing who the sender is, becomes $\frac{1}{10}$, and the corresponding receiver $\frac{1}{9}$ resulting in a total probability of $\frac{1}{90}$.

This does require cooperation from other parties, and as a result, increases the total time required for completion. However, since all parties not involved can continue with their transactions, it does not impact scalability.

6.2.3. Token-centric vs. Account-centric Ledger

Throughout this whole thesis, it has always been assumed that every single party in the network has its own ledger. In this account-centric approach, the ledger belongs to a party, and the current state of a ledger represents the current state of that party's account.

A different approach could be a token-centric ledger. Here every token has its own ledger and is not directly tied to a party. When transferring ownership of an asset, a party does not transfer it from ledger A to ledger B, but instead would transfer ownership of the ledger. This

could, for example, be done by having a field in each block stating the owner. When Alice wants to transfer asset X to Bob, she will create a new block with a public key given by Bob in the owner field, and then sign it with a private key corresponding to the previous owner's public key. Here the public key in the owner-field does not have to be an existing key but can be a new key-pair for every transaction. Prove of ownership, in this case, is done by proving knowledge of the private key.

Even a hybrid version might be interesting, where both the party and the token have their own ledger. Interactions between the physical object and a party are recorded as a transaction between the token's ledger and the party's ledger. This approach might be beneficial in the tokenization of asset management in, for example, a supply chain, where the history of the asset is more important than the entity owning it.

6.2.4. Zero-knowledge Auditing

Another interesting subject for future research would be the development of zero-knowledge auditing. If self-sovereign banking would ever to become accepted as a service not only used by criminals, regulatory compliance is an essential but complicated aspect of it. Privacy regulations and anti-money laundry directives are hard to harmonize. On the one hand, one should be able to engage in transactions without having to reveal information on account balances and previous transactions to the counter-party and witnesses. On the other hand, it should be possible to enforce regulations to prevent money laundering, tax evasion, and financing of terrorism.

One solution would be to use zero-knowledge proofs to prove certain statements about one's transactions. A regulatory body may, for example, require all transactions above a certain threshold to be reported; thus, a party might have to prove that none of the transactions succeed the threshold. Nevertheless, different reports may be necessary to be compliant with different, relevant legal frameworks.

However, making proofs on all kinds of statements might still leak more information than a party is willing to give up. Therefore an interesting concept is to have the role of auditors in a network. Ideally, this should be a role that anybody with the right license should be able to perform. This concept rhymes well with the current way society relies on accountants and notaries to make statements about a company's business without revealing the internal workings or strategies of a company.

A zero-knowledge auditor could audit a (partial) ledger in cooperation with the owner, after which they can create a block stating that the chain has been audited up to that point. Having this approval of an auditor could then be demanded as a prerequisite for specific actions, such as withdrawing money from the system or making large transactions.

6.2.5. Permissioned to Permissionless

Moving from the realm of permissioned networks to permissionless networks introduces many problems. Not only should every node know at any given time know (or have methods to figure out) who are participating in the network (a consensus issue on its own), it also makes assumptions on an upper bound of malicious actors harder to defend.

When everybody is allowed to join the network, nothing would prevent a malicious actor from spawning several other nodes under its control, and let them join the network. Depending on the size, he can either rent out a bunch of servers or use a botnet to flood the network, making it possible to control more than $\frac{1}{3}$ and thus breaking the security assumption of the network.

An often proposed method is using the wealth of a node as voting power. However, this leads to unwanted centralization with the wealthy (The top 1% of the world owns more than 40% of the global wealth). Useful future work could be to research how this work can be adapted to also function in a permissionless.

Bibliography

[1] D. Tapscott, A. Tapscott. “Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World ”, Portfolio, May 2016.

[2] W. Wang, D. T. Hoang, P. Hu, Z. Xiong, D. Niyato, P. Wang, Y. Wen, D. I. Kim, “A Survey on Consensus Mechanisms and MiningStrategy Management in Blockchain Networks” 2018.

[3] J. Pitta, “Requiem for a Bright Idea”. https://www.forbes.com/forbes/1999/1101/6411390a.html, 1999.

[4] S. Micali, and R. L. Rivest. “Micropayments Revisited”. In Preneel B. (eds) Topics in Cryptology — CT-RSA 2002. CT-RSA 2002.

[5] Satoshi Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, 2008

[6] D. Chaum, “Blind signatures for untraceable payments”. In CRYPTO, 1982.

[7] D. Chaum, A. Fiat, and M. Naor. “Untraceable electronic cash”. In CRYPTO, 1990.

[8] https://bitnodes.earn.com/, accessed June 2019.

[9] https://ethernodes.org/, accessed August 2019.

[10] E. Androulaki et al. “Hyperledger Fabric: A Distributed Operating Systemfor Permissioned Blockchains”. arXiv:1801.10228v1, Jan 2018

[11] E. Buchman, J. K. and Z. Milosevic. “The latest gossip on BFT consensus”. https://arxiv.org/abs/1807.04938, September, 2018.

[12] “What is Tendermint?”. https://tendermint.com/docs/introduction/what-is-tendermint.html, accessed June 2018.

[13] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. “The Honey Badger of BFT protocols”. In Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS), pages 31–42, Vienna, Austria, Oct. 2016

[14] D. Schwartz, N. Youngs, A. Britto. “The Ripple Protocol Consensus Algorithm”. ”https://ripple.com/files/ripple_consensus_whitepaper.pdf”, 2014.

[15] G. Bracha. “An asynchronous [(n - 1)/3]-resilient consensus protocol”. In Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC), pages 154–162, 1984.

[16] C. Cachin, S. Tessaro. “Asynchronous Verifiable Information Dispersal” In: Fraigniaud P. (eds) Distributed Computing. DISC 2005. Lecture Notes in Computer Science, vol 3724. Springer, Berlin, Heidelberg

[17] G. Zucman. “Global Wealth Inequality” in The Annual Review of Economics vol 11, 2019

[18] F. Armknecht, G. O. Karame, A. Mandal, F. Youssef, and E. Zenner. “Ripple: Overview and Outlook”, pages 163–180. Springer International Publishing, Cham, 2015.

[19] B. Chase and E. MacBrough. “Analysis of the XRP Ledger consensus protocol”. ArXiv e-prints, February 2018

[20] E. MacBrough. “Cobalt: BFT governance in open networks”. ArXive-prints, February 2018.

[21] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. Comm. of the ACM, 21:558–565,1978

[22] T. Sander and A. Ta-Shma. “Auditable, anonymous electronic cash. In CRYPTO”. 1999

[23] Oscar Williams-Grut, “This Wall Street veteran has raised $107 million to build the ’app store’ of financial services”. www.businessinsider.com/david-rutter-on-r3cevs-plans-for-corda-and-blockchain-after-raising-107-million-2017-6, 2017

[24] R.G. Brown, J. Carlule, I Grigg, M. hearn, “Corda: An Introduction”. https://docs.corda.net/_static/corda-introductory-whitepaper.pdf, 2016

[25] “Cisco Visual Networking Index: Forecast and Trends”, 2017–2022. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html

[26] L. Lamport, R. Shostak, M. Pease, “The Byzantine Generals Problem”. ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, Pages 382-401

[27] L. Lamport. “The Part-Time Parliament”. ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169

[28] M. Castro, B. Liskov, “Practical Byzantine Fault Tolerance”. Third Symposium on Operating Systems Design and Implementation. 1999

[29] Q. Nasir, I. A. Qasse, M. A. Talib, A. B. Nassif, “Performance Analysis of Hyperledger Fabric Platforms”. Hindawi - Security and Communication Networks Volume 2018, Article ID 3976093, 2018.

[30] H. Esselink, L. Hernández, “The use of cash by households in the euro area”. ECB Occasional Paper Series No 201 / November 2017.

[31] M. Young, A. Kate; I. Goldberg, M. Karsten. “Practical Robust Communication in DHTs Tolerating a Byzantine Adversary”. Proceedings - International Conference on Distributed Computing Systems· January 2010.

[32] M.J. Fischer, N.A. Lynch, and M.S. Paterson. “Impossibility of distributed consensus with one faulty process”. Journal of the ACM (JACM) 32.2 (1985): 374-382.

[33] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich. “Algorand: Scaling Byzantine Agreements for Cryptocurrencies”, MIT CSAIL, 2017.

[34] iDeal, “iDEAL-betalingen in 30 maanden verdubbeld naar 2 miljard” https://www.ideal.nl/cms/files/Infographic-iDEAL-betalingen-nov-2018.pdf

[35] E. B. Baker, “Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms”. https://www.nist.gov/publications/guideline-using-cryptographic-standards-federal-government-cryptographic-mechanisms. Special Publication (NIST SP) - SP 800-175B, 2016

[36] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich. “Algorand: Scaling Byzantine Agreements for Cryptocurrencies”. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17). ACM, New York, NY, USA, 51-68. 2017

[37] T. Dinh, J. Wang, G. Chen, R. Liu, B. Ooi, K. Tan. “BLOCKBENCH: A Framework for Analyzing Private Blockchains”. SIGMOD’17. 2017,

[38] E. Buchman. “Tendermint: Byzantine Fault Tolerance in the Age of Blockchains”, 2016.

[39] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. IEEE Computer, Vol. 49, No. 5, pp. 54-63, May 2016. “A Medium-Scale Distributed System for Computer Science. Research: Infrastructure for the Long Term”

[40] “Visa Fact Sheet”. https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf accessed August 2019

[41] G. Maxwell, “https://people.xiph.org/ greg/confidential_values.txt”, referenced April 2018.