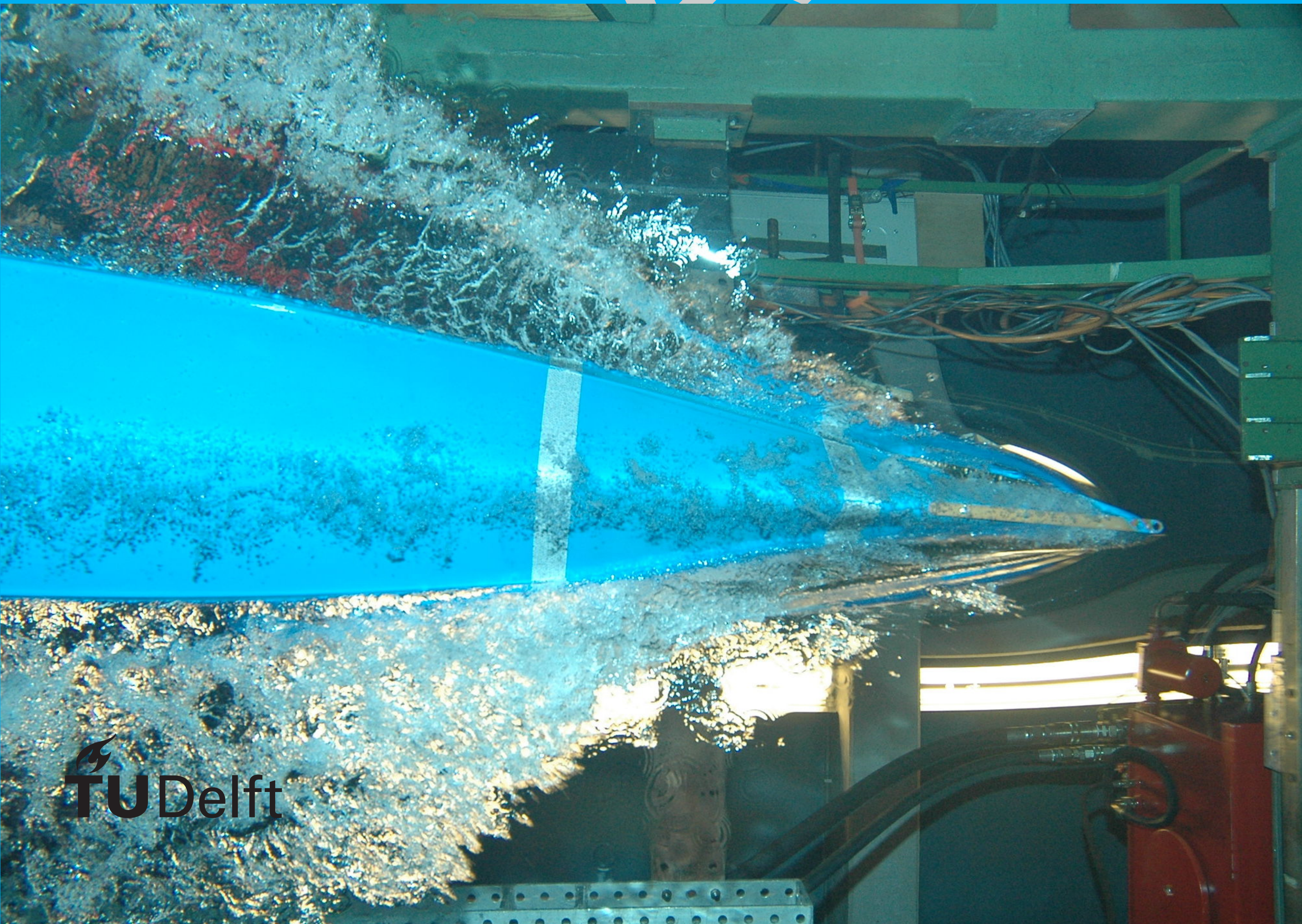
Title

Optional subtitle

J. Random Author

Cover Text
possibly
spanning multiple lines

ISBN 000-00-0000-000-0



TU Delft

Title

Optional subtitle

by

J. Random Author

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday January 1, 2013 at 10:00 AM.

Student number:	1234567	
Project duration:	March 1, 2012 – January 1, 2013	
Thesis committee:	Prof. dr. ir. J. Doe,	TU Delft, supervisor
	Dr. E. L. Brown,	TU Delft
	Ir. A. Aaronson,	Acme Corporation

This thesis is confidential and cannot be made public until December 31, 2013.

TU Delft

Preface

Preface…

J. Random Author
Delft, January 2013

Contents

1 Introduction

When talking about distributed ledgers it is almost obligatory to mention Bitcoin. While the in 2008 released Bitcoin paper[1] sparked a wide interest in cryptographic currencies and distributed ledger technology, it was hardly the first. Already in 1983 Chaum[2] proposed an untraceable digital payment system using blind signatures. A client would create its own banknotes and have them signed by a bank. Since the bills were signed blind a bank would not be able to trace the bill, but a receiver would be ensured that its a valid bill due to the bank signature. In the 1990s and early 2000 several companies (such a DigiCash and Peppercoin) were founded to bring electronic cash out of the cryptographers realm and in to the real world, however none of them succeeded.

One of the issues with digital cash is double spending; a physical €10 bill can only be owned by a single person at the time, and can only ever change from owner not replicate itself. With digital cash however, there is no guarantee that after transferring the money the buyer actually deletes the digital €10. This is where Bitcoin out-shined the competition, where some solutions relied on calling the issuing bank to verify the coin is still valid[3] or having each bank host a public database with all currently valid coins[8], Bitcoin publishes all transactions on a public ledger. But in contrast to a single bank being responsible for this ledger, it is a shared responsibility. Transactions are bundled in blocks and this block is completed by a pointer to the previous block and a solution to a cryptographic puzzle. Having solved this cryptographic puzzle entitles to Bitcoins, incentivizing people hosting and maintaining the ledger. The processes of solving this puzzle is called 'mining'.

By artificially keeping the time it takes to 'mine' a block high, the probability of multiple parties creating a block at the same time is kept low. If this does happen other parties can arbitrary pick one block they consider valid and after a while one of those blocks will have a successor. By a simple rule stating that the longest chain is the valid one eventually consensus will be reached on which blocks and therefore which transactions are valid.

This eventual agreement upon 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 tries to fix the double spending problem by implying a total order on all the transactions. While it seems that global consensus does a good job at preventing the double spending problem, it does create some new issues.

A characteristic requirement 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 and unless an algorithm with a run-time complexity of $O(1)$ is used, the capacity of such a system will only decrease with the addition of more nodes.

Another side-effect of global consensus is that by implying total order on all transactions, all the nodes should posses information about all the transactions. At the time of writing there were 10412 known Bitcoin nodes each storing the complete blockchain of 201 GB, a total of 2 Petabyte of data stored.

But even when assuming a magical consensus algorithm, that reaches consensus without any added latency, most distributed ledgers employ a single block chain resulting in multiple parties working on a single 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 will explore the possibility of creating a secure distributed ledger while circumventing the use of global consensus.

This exploration will start in chapter 2 with a problem description. In chapter 3 related work is discussed. In chapter 4 etc etc etc.

2
Problem description

Early on in their lives computer scientists made aware of all the evil that global variables bring. Yet here we are, the R3 consortium (receiver of the largest single investment for a distributed ledger technology company at $107 million[9]) claims in [10] that its 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 a dispute, one can simply take a look at this globally agreed upon ledger with all agreements and settle the dispute.

However, from a computer science perspective requiring a global data structures introduces a lot of difficulties, of which some are even proven impossible to solve. To see why these problems exist it is important to have a better understanding of distributed systems and consensus algorithms.

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

Having a single central server with all the data in the world does not only seem a bad idea regarding availability and security, but attempting to serve the 122.000 petabyte per month of IP traffic as estimated for the internet in 2017[11] from a single server seems futile. By distributing the load across multiple server this performance can be achieved.

The reliability and availability arise from the notion that, in a well-designed system, the system can keep functioning even when one or more computers fail.

As a result of these favorable properties distributed ledger technology has sparked a wide interest in enterprises and financial institutions. However, just as with large groups of people working together, disagreement can occur (be it due too malformed information, delayed information, or malicious behavior), and requires solving before proceeding.

Reaching consensus is an important fundamental problem in computer science as it emerges in a lot of applications of distributed systems such as 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 in the real world processors 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 as permanent or transient faults. Transient faults occur temporary, for example corruption of memory due too 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 of a processor will not corrupt messages; A message is either sent entirely or not at all.
2. *Byzantine failure model*: In this model everything may happen. A processor is allowed to randomly stop sending or receiving messages, or arbitrary decide to send messages with any content to other processors in the system. This also include 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 model is often used 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 also misbehaving nodes due to malicious behavior are to be considered. Since byzantine failure tolerant algorithms also include tolerance against knowingly and willingly malicious behavior they give a higher assurance regarding security, but is also more difficult to complete. For some cases byzantine fault tolerance can even be impossible. A good example of this is the three generals problem first publish in [12] by Lamport, Shostak and Pease.

Imagine three generals, Alice, Bob and Charles, of which one is a traitor. Each have there army surrounding a city, but to be successfully they must either attack together, or not attack at all. Say that Bob proposes a plan and informs Alice and Charles about the plan, however Bob is malicious and tells Alice to attack but Charles to retreat. When Alice then confirms with Charles what to do she gets conflicting information.

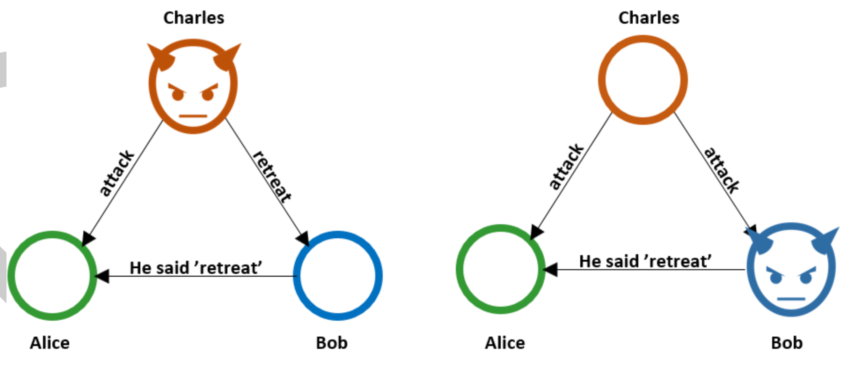


Figure 2.1: On the left a situation where Charles is the malicious general, On the right Bob is the malicious general. From Alice's perspective the situations are identical, making it impossible for Alice to make a correct decision.

Now imagine the same scene, however, not Charles but Bob is the malicious actor. When Alice asks Bob for confirmation, he lies and says that Charles told to retreat. From Alice her perspective it is impossible to distinguish between the two situations, and can therefor not make the correct decision. Where the paper shows 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 at most m parties are considered malicious when designing distributed algorithms.

Traditional consensus algorithms Since consensus is a fundamental problem in computer science a tremendous amount of research has been performed in this area. One of the first to offer a formal solution and safety proofs was the Paxos-family first published in [13] in 1989. 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 where multi-value Paxos requires to reach agreement on multiple values and as well as an order of those values. While Paxos does gives guarantees on certain properties it is impractical due to its complexity, even the practical Byzantine fault tolerance variant described in[14] comes with a message complexity of $O(n^2)$. Where this complexity isn't 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 9.000 known nodes, a message complexity of $O(n^2)$ directly renders PBFT unfeasible.

Global Consensus Distributed ledgers such as Bitcoin and Ethereum have employed a different method for reaching consensus; proof of work. The specific proof of work mechaisnm 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 Bitcoint 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 of 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 ($P = 0.5^{no.\ bits}$). The difficulty is dynamically changed to compensate for the increase in computational power of the network, and is kept to have an average of 6 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 finding consecutive blocks simultaneously decreases exponentially, one chain of blocks will eventually become longer and will be accepted as the truth and all other blocks will be discarded.

While this consensus algorithm is proven to scale up to 10.000 nodes it has some shortcomings. Firstly, the time it takes for a block is invariant of the number of users on the network (it always takes about 10 minutes for a block the be created) and since every block has a maximum number of transactions, the network's throughput is limited to 7 transactions per second. Secondly, all miners would like to be the first to find the block and will use all of there 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 this shortcomings, companies like IBM, R3, and Google research higher throughput and more energy efficient methods of reaching consensus.

While it would be foolish to question the capabilities of companies such as IBM and Google on developing new consensus algorithms, one might wonder if global consensus merely a solution, not the solution. The goal of global consensus in distributed ledgers is to prevent invalid transactions from taking place.

To determine the validity 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 transaction must have a certain order.
3. *legality/validity*: based on the current state of the ledger, does the entry specify a legal/-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 independent 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.

### Hier moet even een bruggetje naar Trustchain

Trustchain Trustchain is a distributed pair-wise ledger developed at Delft university of Technology. Trustchain distinguishes itself from traditional DLTs by not having a single ledger containing all transactions, but having a single 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. This jointly created block will then be appended to the chain of all involved parties through a hash and irrevocably entangles ledgers of all involved parties.

each block points to a previous block which means that each block effectively is a record of an happened-before relation as described by Lamport in [7]. Due to these pointers and the transitivity of the happened-before relationship 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 transactions that affect a party that are not recorded on that party's ledger can exist.

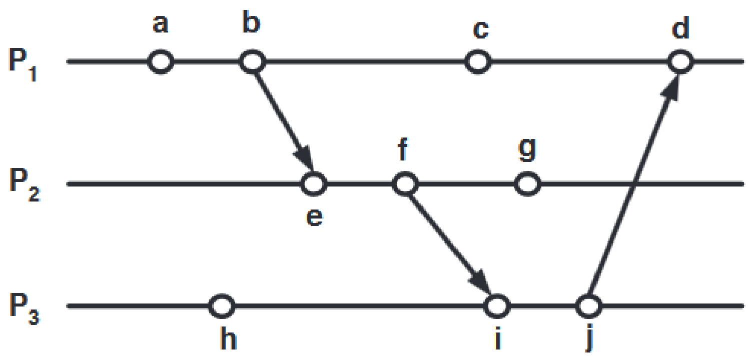


Figure 2.2: An example of the happened-before relation. The horizontal lines represent ledgers, and the The events a, c, and h do not involve P1, P2, or P3 as counter-party so the arrows are omitted for clarity.

In figure 2.2 an example of the happened-before relation is shown. When thinking of the arrows as beneficial monetary transaction in the direction of the arrow, it can be easily reasoned that causal ordering of transactions are already sufficient for correct execution of these transactions. To execute transaction $j \rightarrow d$ only the order $i \rightarrow j$, $f \rightarrow i$, $e \rightarrow f$, $b \rightarrow e$, and $a \rightarrow b$ is relevant to asses the validity. The exact timing with respects to c and e, f, g, i, and j is irrelevant, knowing that $c \rightarrow d$ and $b \rightarrow c$ is enough to assess the current state of P_1's ledger.

By having this implicit causal transaction ordering, the necessity for total order can be dropped. Since the Trustchain data structure is inherently causally ordered, and a single ledger already describes the total current state of the corresponding party, the remaining reason for global consensus would only be to prevent invalid (potentially malicious) transactions. When invalid transactions can be prevented without global consensus, the requirement for a single data structure and a majority vote per transaction can be dropped. As a result, multiple parties (that do not depend on, or interact with each other) can be transact concurrently with each other, enabling a highly scalable DLT. 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 will explore the possibility of creating a secure distributed ledger while circumventing the use of global consensus. This leads to the question being answered in this thesis:

“How to achieve a secure distributed ledger without global consensus?”

3 Fair Witness Selection Protocol

This chapter focuses on 'Fair witness selection protocol, which is the driving factor behind the security of our distributed ledger. The core concept of the proposed solution is to have witnesses sign of on valid transactions. The question than arises, how to select those witnesses: On one hand they should be selected random; nodes should not have the ability to steer the nodes select to do the witnessing. As this would give malicious actors the ability to only select malicious witnesses. On the other hand the witnesses selection should be verifiable; for any given statement it should be easy to verify that the involved witnesses are selected according to the rules.

3.1. Formalization

We begin the description of the systems 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 this to the network.
- **Transaction** An agreement between two ore more parties, including a publicly verifiable signature from every party and pointers to the last blocks of the transactors unique chain,.
- **Witnesses** A node that is not involved in the transaction, but is chosen to oversee the the transaction and approve it in case of a valid transaction.
- **Block** A block is a data structure containing the transaction, the witness' 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). when talking about the chain of a specific party, the ordered set of all blocks co-created by that party is meant.

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) randomly send or refuse to send messages.

3.2. System model

FWSP makes certain assumptions on the system for it to work. 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 termination impossibility in asynchronous systems as described by Fischer, Lynch, and Paterson[**?**]. We assume that every node node has contact information, earlier blocks, and the signature verification key for every node (or means to acquire this); be it through a predetermined look-up table, or a byzantine-fault tolerant DHT[**?**]. As proven in [12] there are no solutions for consensus algorithms with more than $\lceil \frac{n}{3} \rceil - 1$ malicious nodes, therefor we assume this as the upper-bound. It is assumed that the unique identifiers are in the same domain as the H().

It is assumed that:

- nonfaulty nodes only propose valid transactions, will respond timely (i.e. before message time-out) and will sign every valid transaction.
- All malicious nodes collude, only propose invalid transactions, and will only sign transactions from malicious nodes.

Refusing to sign any valid transaction simulates a denial-of-service attack on honest nodes. It assumed that transactions are agreed uppon and signed by the involved parties before initiating this protocol.

3.3. Fair Witness Selection Protocol

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

Now the question arises, how to pick witnesses in a random but verifiable way. Random because no party should be able to manufacture a transaction in such a way he has control over the selected witnesses. Verifiable as anyone in the network should be able to verify that the witnesses were selected according to the algorithm.

The design goals are as following:

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

3.3.1. Definition

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

Let $Sign(\{0,1\}^r, sk) \to \{0,1\}^n$ be a secure signing function, and $Ver(\{0,1\}^n, pk) \to \{true,\ false\}$ the corresponding verification function.

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

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

Due to the uniform distribution of the hash function, node are effectively selected at random. Furthermore, the pre-image Resistance of the hash ensures that its difficult to find a transaction that will result in a given ID. There is a non-zero possibility that the selected node is malicious, however the probability that this and every successive witness is malicious decreases exponentially. A larger minimal witnesses set result in a higher security parameter. In section 3.3.2 a mathematical proof of this security parameter, along with a the minimum witness set for a variety of assumptions and requirements are given.

A trivial attack against this selection process would be to keep increasing i until enough malicious witnesses are found. To combat this also the i used for determining the witness is posted, along with the 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.

An honest node can create a valid block by selecting his witness using equation 3.1 and request those witnesses to sign the transaction. Again, there is the non-zero possibility that less than k honest nodes are among the witnesses. however, since by definitions $|H|$ is at least twice $|M|$ the probability of an honest majority exponentially increases with every increment in the witness set.

Block creation The creation of the block starts by verifying the validity of the block. if the block is valid the node calculates the hash for the block with a 0 append, and selects the node who's ID is the closest but smaller or equal to the hash and sends him the chosen ID, transactions and 0. The node repeats this process now with 1, 2 ... up to s, (the minimum number of required witnesses) instead of 0.

Every time a node receives a $\perp$ 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.

From here on the transaction is concatenated with the signatures, and than finalized by concatenating the hash of the transaction and signatures to create the final block.

Block signing On receiving a signature request, the receiving node first verifies that the transaction is valid and he indeed is the required witness. If this is not the case, the node will reply with $\perp$, otherwise the node will append the given i to the transaction and sign it. By appending the i the node prevents any malicious actor to make false claims about the number of nodes selected as witnesses by modifying i after having received the signatures. The node's ID concatenated with signatures and i are then send back to the receiver.

Block verification Block verification is a non-interactive process. The verification is straight forward, and is started by checking the hash of the block is correct, and the number of valid signatures is equal to or greater than the minimum required. Next the largest witness set is such that $|validSigantures| \geq \lfloor \frac{2|W|}{3} \rfloor + 1$ still holds, is calculated to ensure no i larger than maximum allowed witness set. For each witness it is verified that they were selected according to protocol, and that the signature is valid. If all of these tests pass the block is valid, otherwise the block is considered invalid.

3.3.2. Correctness

Every valid transaction must be accepted in finite time If an honest node proposes a block containing a valid transaction, but less than $\lfloor \frac{2|W|}{3} \rfloor + 1$ nodes reply with a valid signature (due to malicious behavior or due to faulty behavior) the node will increase the witness set. It will do so until it has collected enough valid signatures or until $W = N$.

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

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

A transaction 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, which gives:

$$v = |H|$$

Algorithm 1: createBlock(transaction, s)

```
$t \leftarrow transaction, validSignatures \leftarrow \emptyset$
if $V(t)$ then
    for $i \leftarrow 0$ to $s$ do
        $w \leftarrow \sup\{z \in N : z \leq H(t||i)\}$
        requestSignature(w, t, i)
        $W \leftarrow W \cup w$
    while $|validSignatures| < k$ do
        if requestSignature() time-out then
            $i \leftarrow i + 1$
            $w \leftarrow sup\{z \in N : z \leq H(t||i)\}$
            $W \leftarrow W \cup w$
            send requestSignature(w, t, i)
    $block \leftarrow t$
    foreach $s \in validSignatures$ do
        $block \leftarrow block||s$
    return $block||H(block)$
```

Algorithm 2: on receive requestSignature(t, i)

```
$t \leftarrow transaction, p \leftarrow nodeID$
$w \leftarrow \sup\{z \in N : z \leq H(t||i)\}$
if $V(t)$ and $w = p$ then
    $s \leftarrow Sign(t||p||i, sk)$
    reply $p||s||i$
else
    reply $\perp$
```

Algorithm 3: verifySignatures(block, s)

```
$t, signatures, hash \leftarrow interpret(block)$
if $|signatures| < s \vee hash \neq H(block)$ then
    return $\perp$
$w_{max} \leftarrow \lceil |signatures| \times \frac{3}{2} \rceil - 1$
if $\{\exists s \in signatures : i > w_{max}\}$ then
    return $\perp$
foreach $e \in signatures$ do
    $w, Sign, i \leftarrow interpret(e)$
    $w_{expected} \leftarrow sup\{z \in N : z \leq H(t||i)\}$
    if not $(w = W_{expected}$ and $Ver(Sign, w_{pk})$ then
        return $\perp$
```

Therefor a block is valid if

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

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

An invalid transaction will only be accepted with a negligible probability For an invalid transaction to succeed at least k nodes within the witness set should we malicious.

The probability of a portion of a subset, drawn at random from a finite population, being malicious can be modelled with the hypergeometric distribution.

For an actor to have a malicious transaction accepted requires at least k malicious nodes in the witness set. To model this using the Hypergeometric distribution we consider drawing a malicious node is a success.

Let X be a random variable corresponding to the amount of malicious actors in the witness set. The probability mass function of X is 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 malicious 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 signatures will pass an invalid transaction, but any number greater than k will also succeed. The probability of a malicious transaction succeed 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:

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

3.3.3. Liveliness

Since every valid transaction eventually will succeed, every party involved in a valid transaction will eventually be able to proceed. Since signing and creating transactions are two separate procedures that have no dependencies on each other, transactions can be signed during a transactions, transactions can be started while signing, and two signature requests can be fulfilled concurrently.

3.3.4. Security

The security of the protocol relies heavily on the minimum required number of witnesses. If a single witness would be allowed the probability of a malicious transaction succeeding is equal to 0. A larger witness group results in a lower probability, the exact number of witnesses depends on the allowable probability of malicious transactions and the number of malicious nodes in the network.

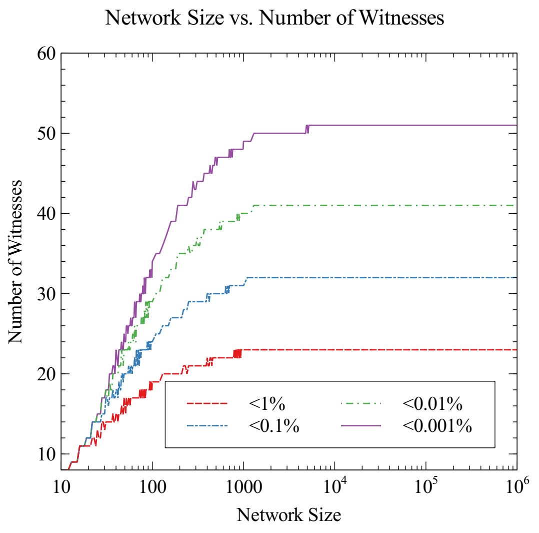


Figure 3.1: Minimum number of witnesses as a function of the network size. The malicious nodes make up 1/3 of the total network. The number of required witnesses are given for $< 1\%$, $< 0.1\%$, $< 0.01\%$, and $< 0.001\%$ chance of malicious transactions succeeding.

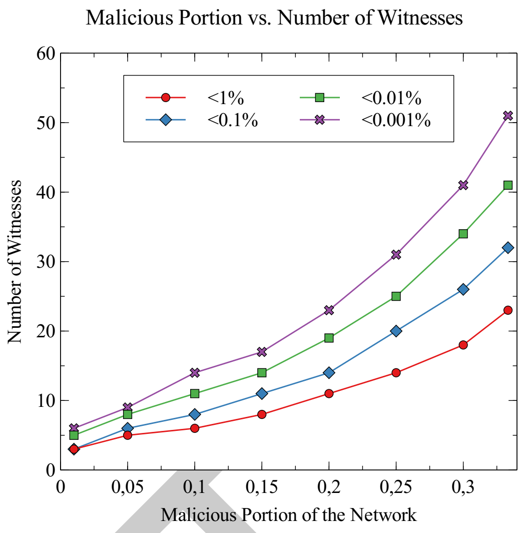


Figure 3.2: Minimum number of witnesses as a function of the malicious portion of the total network. The number of required witnesses are given for $< 1\%$, $< 0.1\%$, $< 0.01\%$, and $< 0.001\%$ chance of malicious transactions succeeding.

In figure 3.2 the minimum number of required witnesses is given, where X-axis is the portion of the malicious portion of 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 probability of 0 (See section 3.3.2 for proof).

When assuming a the number of malicious nodes to be $1/5^{th}$ of the complete network (Similar to Algorand [**?**]) the network only requires 23 witnesses to guarantee $< 0.001\%$ change of malicious transactions. To guarantee a probability of$< 5 \times 10^{-9}$ or less 38 witnesses are required, whereas Algorand requires 2.000 to achieve the same level of security (for a network of 10.000 nodes).

3.3.5. Enhanced security

While $< 0.001\%$ for a malicious transaction to succeed is a small change, 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 stating that not only the current transaction should be verified, but also a give number of former transactions. In practice this comes down to not only sending a singlet block to the witness for the previous-hash pointer, but send the last n blocks for both determining the previous-hash pointer and verifying the validity of the transactions.

A malicious party that successfully performed an invalid transaction will not be able to create a new block that doesn't have majority of malicious witnesses. If this malicious party would ever like to reap the seeds of his crime, he has to hide the malicious transaction behind n 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 5 blocks. If the node would ever want to perform transactions involving non-malicious peers, the malicious transaction should be succeeded by at least 5 valid transactions. However, these valid transaction will not be signed by honest witnesses as they would be required the last 5 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 suddenly becomes p^6. When we assume an initial probability of $< 0.001\%$ this gets increased to $< 1 \times 10^{-18}\%$

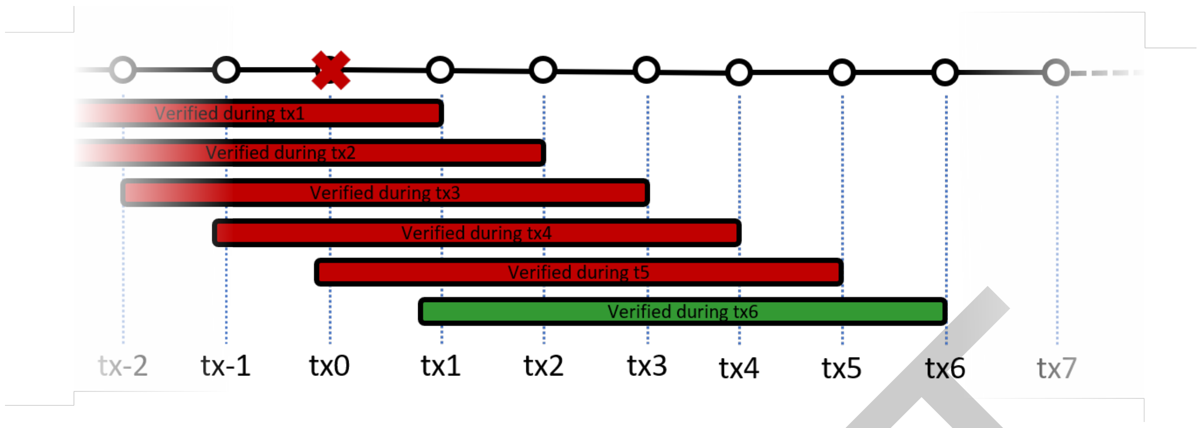


Figure 3.3: Graphical representation of a malicious transaction being followed by valid transactions. In this example $tx0$ is the malicious transaction, $n = 5$ a red bar indicates a verification that would fail, where a green bar indicates a success.

This case can be generalized to n blocks with an initial probability of p. If the probability of performing a malicious transaction is p, and n valid transactions need to be signed by an dishonest witness set, the probability of successfully performing an attack is:

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

The security of a system is only as strong as its weakest link. The National Institute for Standards and Technologies, states the highest approved security strengths for federal applications is 256-bits strength[17]. When following these guidelines the probability should be lower than guessing the private key; $< \frac{1}{2^{256}}$.

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 256 bit security can be seen.

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

Table 3.1: 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

In table 3.2 the number of previous transaction that need to be verified to reach 128 bit equivalent is given.

3.4. FWSP Against common attacks

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

$P_Malicious =$	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 3.2: 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

Inadmissible transactions Inadmissible transactions are transactions that are well-formed but are not valid with respect to the networks rules. Examples of these would be double spending attacks, but also unauthorized creation or destruction of assets. These attacks, however dangerous they are to the network, are rather trivial to protect against. If all the requirements that constitute a valid transaction are embedded in a function that is used by fwsp, and a block is only accepted if it comes with a given number of signatures (in accordance with fwsp). The probability of a inadmissable transaction being accepted can be negligibly small, dependent on the fwsp parameters.

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

To prevent the hiding of blocks the fair witness selection protocol can be used to create a reliable way of block dissemination and replication. To do so not use the hash of the current transaction, but of the previous block. When combining this with an network imposed rule that witnesses are not only supposed to witness the transactions, but also store the block, the probability of successfully hiding a block becomes even lower than that of passing a inadmissible transaction. Where as passing 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-reputability 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 $|W|$. Which results 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 which $1/5^{th}$ is malicious a minimum witness set of 23 nodes will result in an inadmissible transaction having a $< 0.001\%$ chance of being accepted, where it only has a $< 1 \times 10^-16\%$ chance of successfully hiding a block.

When a node is presented a block of which it is claimed it is the last block, it can now be easily verified. Using formula 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.

3.5. FWSP input

What data is used as an input for the fair witness selection protocol has a great impact on the security. If 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 unfeasible. However, when relying on the technique described in 3.3.5 to increase security this is not sufficient.

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

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

doesn't 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 down side 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. In this method additional care must be taken, to ensure that all signatures given by witnesses on a previous transaction. 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 21 valid signatures, he only needs 14 or more to be accepted as valid. The amount of unique valid combinations for this given example rises to:

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

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

4 Proof of Concept: Self-Sovereign Banking

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 a higher reliability due to the lack of single point of failures, while having lower operational costs.

The payment network is hosted by everyone who's taking part in it, and enables participants to transfer funds to each other similar to online banking. For this design a homogeneous network is chosen, so that every node who 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', which can only initiate but not witness transactions, and 'full-clients' which can both witness and initiate transactions.

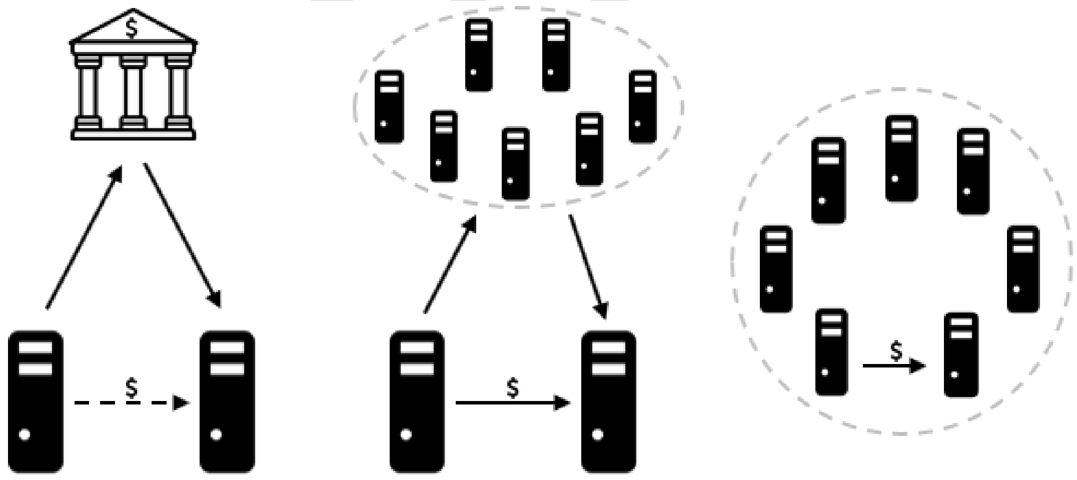


Figure 4.1: Graphical representation of different topologies of payment networks in which Alice wants to transfer funds to Bob. Nodes in the circle are full-clients. Left) A traditional payment network; Alice informs the bank about the transaction, which will route the money to Bob. Centre) A heterogeneous network where Alice and Bob are light-clients; Alice directly transfers Funds to Bob, but he can't 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.

To ease the design and enable code re-usability the software architecture is of modular nature. The design is split up in four distinct pieces: User Interface, Business Logic, Trustchain core, and storage.

4.1. Software architecture

4.2. User Interface

The user interface is designed as a web-app. Bootstrap, a popular open-source front-end framework, is used to generate a responsive website which can be operated on both desktops and mobile devices.

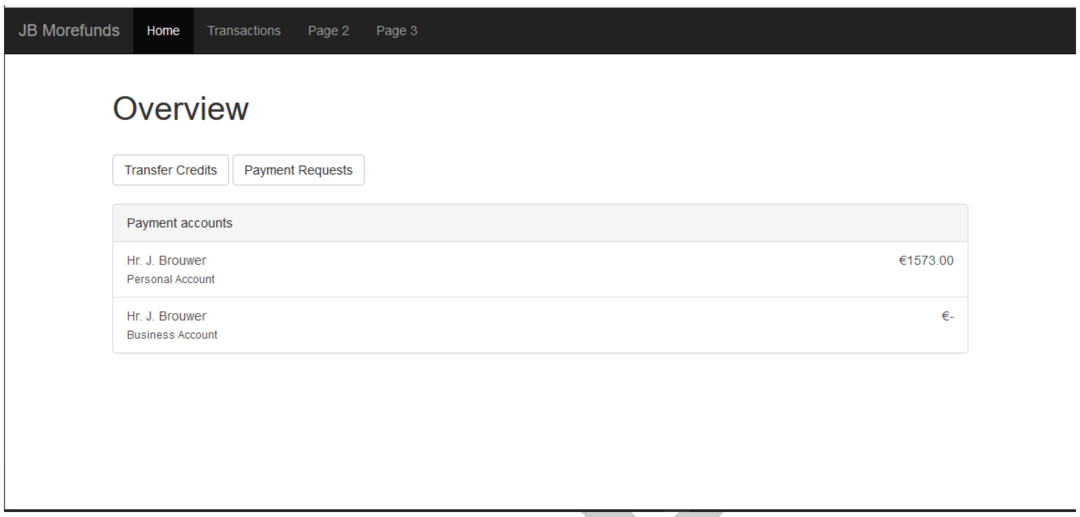


Figure 4.2: The home screen of the website, showing an overview of the bank accounts and the balances.

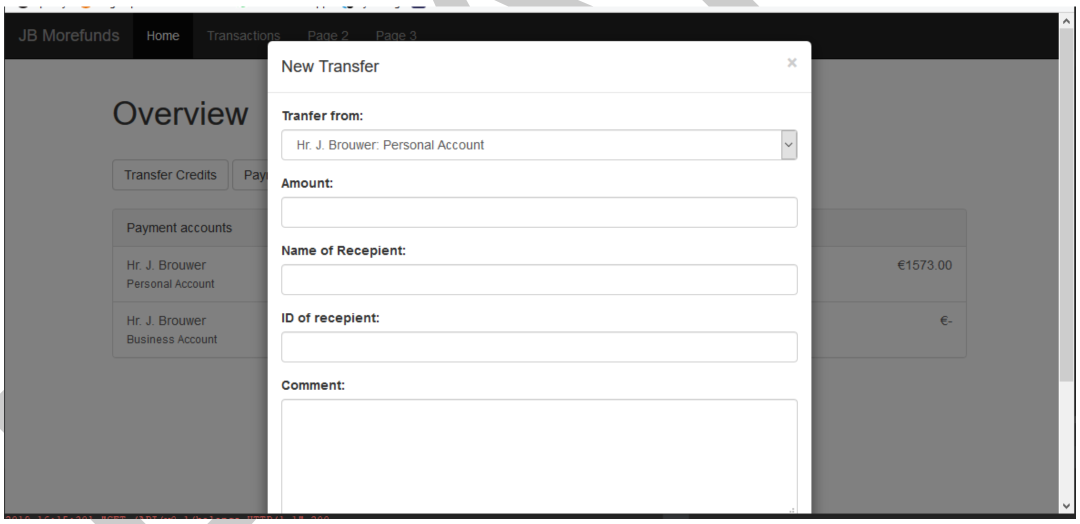
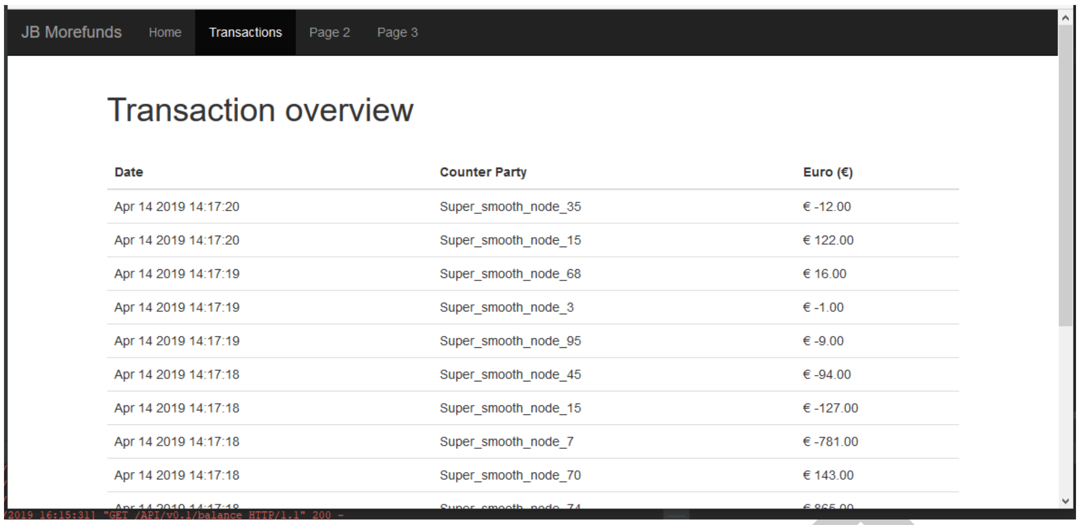


Figure 4.3: The view from which the user can initiate a new transaction.

The front-end uses http-calls to query the REST-api hosted by the application. the REST endpoint currently supports three calls: 'Balance', 'Transactions', and 'StartTransaction'.

'Balance' takes no arguments and returns the current balance. 'Transaction' takes two optional arguments 'Start' and 'length', and returns the time stamp, counter party and associated transaction value. When the 'start' argument is supplied the back-end will return the 25 transactions that directly came before the transaction referenced in start, if nothing is supplied the 25 most recent transaction are returned. 'length' determines the number of results returned, if not supplied it will return 25 transactions.

StartTransaction takes the recipient's ID and value as arguments and returns the transaction if successful, or an error in case of one.



Transaction overview

Date	Counter Party	Euro (€)
Apr 14 2019 14:17:20	Super_smooth_node_35	€ -12.00
Apr 14 2019 14:17:20	Super_smooth_node_15	€ 122.00
Apr 14 2019 14:17:19	Super_smooth_node_68	€ 16.00
Apr 14 2019 14:17:19	Super_smooth_node_3	€ -1.00
Apr 14 2019 14:17:19	Super_smooth_node_95	€ -9.00
Apr 14 2019 14:17:18	Super_smooth_node_45	€ -94.00
Apr 14 2019 14:17:18	Super_smooth_node_15	€ -127.00
Apr 14 2019 14:17:18	Super_smooth_node_7	€ -781.00
Apr 14 2019 14:17:18	Super_smooth_node_70	€ 143.00
Apr 14 2019 14:17:18	Super_smooth_node_74	€ 865.00

```
2019 16:15:31] "GET /API/v0.1/balance HTTP/1.1" 200 -
```

Figure 4.4: An overview showing the most recent transactions.

4.3. Business Logic

4.4. Trustchain Core

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

4.4.1. Half-blocks

Tribler's current Trustchain implementation uses half-blocks to enable concurrent transactions, to deal with slower peers. When only allowing one transaction at a time a delay in a single peer can prevent the creation of further blocks. The half-blocks essentially are trade-proposals which are directly added to the proposer's chain. If the counter-party chooses to accept this trade, he can then create the other half of the block which will reference the first half.

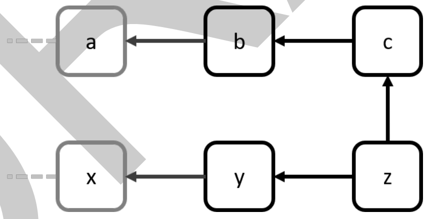


Figure 4.5: Graphical representation of two individual chains, linked together through a common transaction.

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.

Since the creation of the half-block doesn't depend on the counter-party 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 counter-parties, it only delays the outings 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 in a sensible way, 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 basically becomes worthless: Unless the complete chain is analyzed to determine which portion of transactions that are valid, there is no way of knowing the actual current state. (This is the current implementation in Tribler with respect to the 'running total'.)

- Update after receiving the corresponding counterpart: When including the transaction in the next block after receiving a signed counterpart, transactions becomes 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's 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 doesn't exist).

  The issue of seemingly random changes in the state could also be solved by implementing a new type of block, signed by both parties with ca copy of the transaction that is completed. This makes the state-change in a logically and easy to verify manner, however the counter-part is then redundant as the information is already stored in the block that signals the completion of the transaction.

In both cases it 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. Doing so might slow down the block creation but improves block verification as transaction 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.

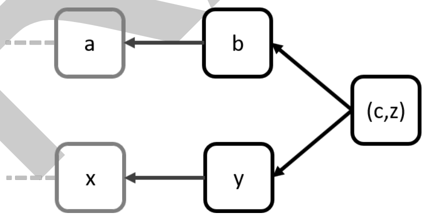


Figure 4.6: Graphical representation of a jointly created block.

If no mutations to the chain is 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 his on timeline. Which opens up a whole variety of potential problems, look for example figure below:

In figure 4.7 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 transaction in which they lose 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. The total sum of tokens becomes negative, which violates the requirements of always having a positive balance (i.e. a double spending attack).

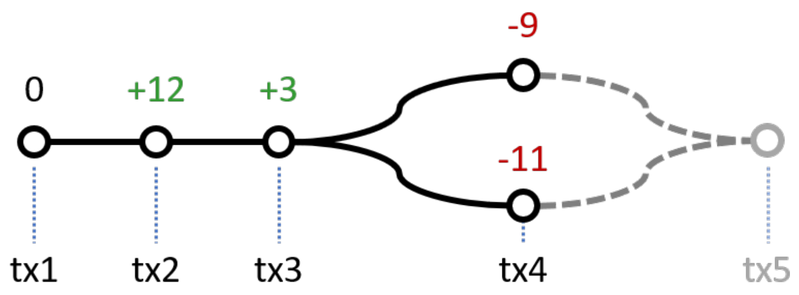


Figure 4.7: Schematic overview of transaction-timeline with a fork due to concurrency.

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

The above-mentioned problem 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're 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 spend 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.

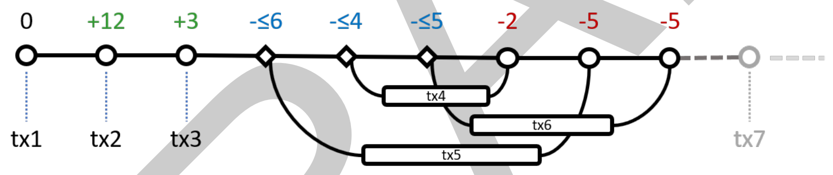


Figure 4.8: Schematic overview with explicit forking with three concurrent transactions.

The example in figure 4.8 uses the same concept as figure 3, however now it creates a block to explicitly indicate a fork in the timeline. 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 greatly increases the verifiability of a chain since the maximum number of blocks that need to be checked before ensuring there are no open forks = $X \times 2$.

4.4.2. Serialization

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

Adding, removing, or changing a field's type would require changing the database schema and converting every block from the old to the new form. This would not only require a lot of computation where each peer can have thousands of blocks, it also breaks backwards 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

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 4.9: Database schema used in the current Implementation in Tribler.

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

Listing 4.1: A transaction taken from the current Tribler implementation

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 have offer some flexibility the transaction data is stored in plain-text using JSON notation to store the actual contents of the transaction. While this does offers the option to change the format of a transaction without breaking backwards compatibility it comes at the cost of size, the transaction in fig 4.2 for examples 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.

JSON JSON is an open-standard file format that stores data in 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 and communcates using objects described in a key-value pairs. The benefit from this is that no predetermined format is required, as any object can be simply added to the messages, offering both eas of use and flexbility. Since JSON is widely and used and uses plain-text most mainstream programming langauges either have built-in support of have wel written libraries JSON is very portable accross both langauges and platforms. A down side of JSON is that everything is stored using plain-text. For storing string this poses not 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 containt 767 signifigcant 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 both simple and performant and be both platform and langauge agnostic. Protobuf is used by first defining a schema similar to defining structs in C in a '.proto' file. This protofile 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 serialized in a certain language can be de-serialized in any other language, offering great portability. While Google claims that Protobuf messages are backwards-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 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 requires writing parses for every language that needs to be supported. When needing to support dynamic length data types such arrays a custom format becomes a bit more complicated. Flexibility is also very limited as changing the format would require updates to all instances.

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 4.4.2 both Python's pickle and Protobuf seem good contenders. The pickling method outshines Protobuf when it comes to simplicity because it doesn't require additional tools or a different language to define schema's. But Protobuf's platform and language agnostic property opens the door to implementing smartphone or web-apps versions or clients, therefor Protobuf is considerd the best for serialization.

4.4.3. Block format

The block and transactions are split up in to two separate Protobuf messages. The reason for this is that a transaction needs to be signed before it can be send to the witnesses. Protobuf uses the conecpt of 'varints' which gives the posibility to store an 64bit integer with less bytes if the value could be represented with less 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 do not require the shortest version to 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 do not prevent the message to be interpreted on any other platform, it does create a problem with regarding to the signatures, a single bit difference renders a signature invalid.

4.5. Storage

```
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 4.2: excerpt from the '.protofile' defining the transaction and block format

5

Experiments & results

Bibliography

[1] Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, 2008

[2] D. Chaum, Blind signatures for untraceable payments. In CRYPTO, 1982.

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

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

[5] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. “The Honey Badger of BFT protocols”. In Proceedings of the23rd ACM Conference on Computer and Communications Security (CCS), pages 31–42, Vienna, Austria, Oct. 2016

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

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

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

[9] 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

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

[11] ”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

[12] L. Lamport, R. Shostak, M. Pease, ”The Byzantine Generals Problem”. 1982

[13] L. Lamport, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169

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

[15] H. Esselink, L. Hernández, ”The use of cash by households in the euro area”. 2017

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

[17] 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