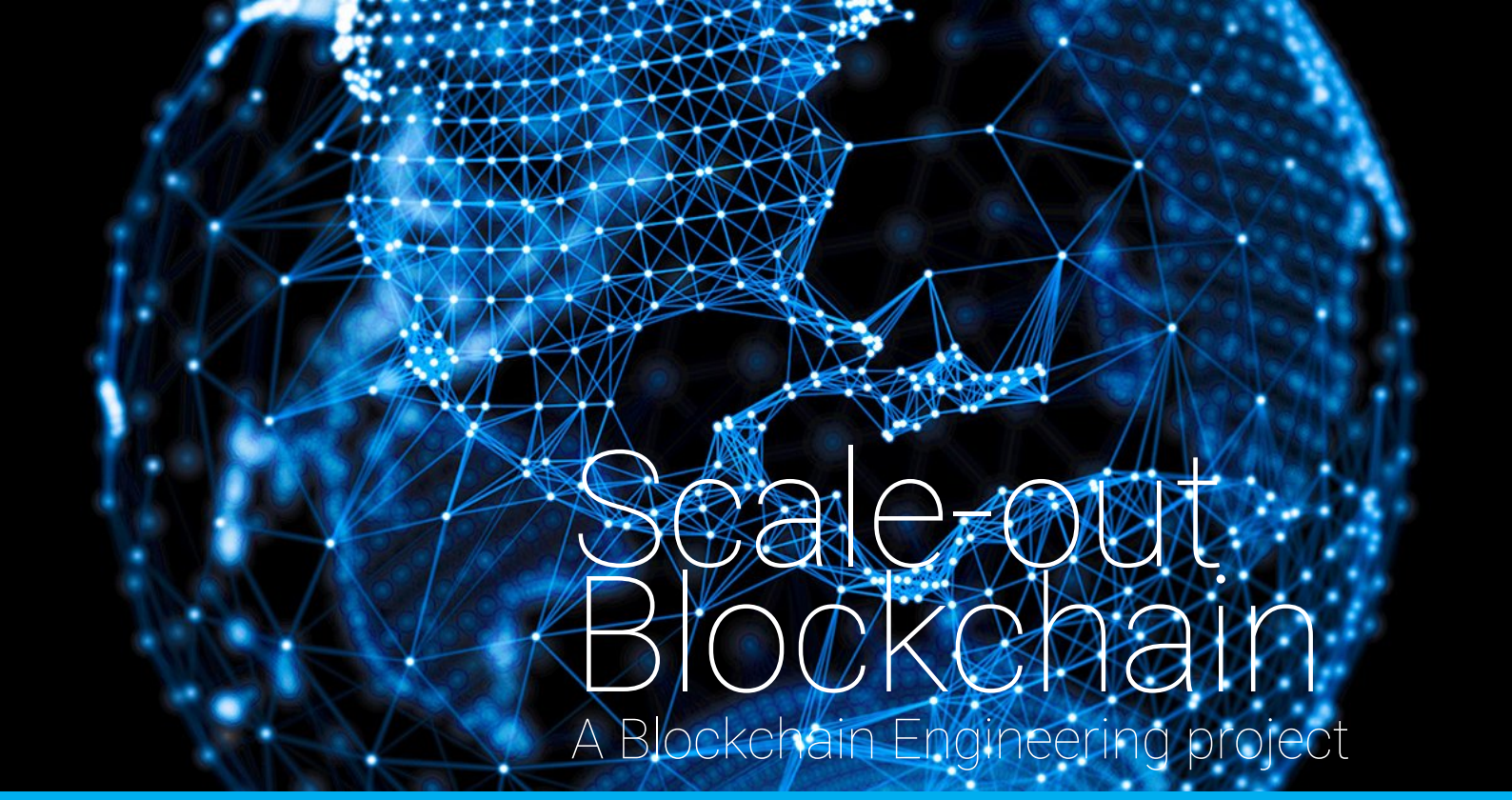

Scale-out Blockchain

A Blockchain Engineering project

Taico Aerts
Chiel Bruin
Bart de Jonge
Karol Jurasinski
Alex Morais

Technische Universiteit Delft

TU Delft

Scale-out Blockchain

A Blockchain Engineering project

by

Taico Aerts
Chiel Bruin
Bart de Jonge
Karol Jurasinski
Alex Morais

Taico Aerts	4345797
Chiel Bruin	4368436
Bart de Jonge	4392256
Karol Jurasinski	4748069
Alex Morais	4747240
Supervisors:	Z. Ren
	Z. Erkin

An electronic version of this report is available at
`github.com/bartdejonge1996/ScaleOutDistributedLedger`.

TU Delft

Contents

1 Introduction

At the moment of writing (early 2018) blockchain is a hot topic. Recently this lead to the exchange rates for Bitcoin and other cryptocurrencies soaring to enormous heights[1]. As a result, the shortcomings of these systems became even more clear. Firstly, the throughput of transactions is low, but an even more important issue is the enormous power usage by all the miners. One way to eliminate the need for miners in blockchain systems is to switch from a proof of work/stake to a new way of reaching consensus. At this moment, there are algorithms available that create Byzantine-fault-tolerant (BFT) systems, which allow all the nodes participating in the blockchain to reach an agreement. An example of such an algorithm is PBFT by Castro et al. [1]. However, a problem that still persists is that, traditionally, these systems have a communication cost per transaction (CCPT) that is of order $O(N^2)$[1]. Some algorithms may also improve this to be lower bounded by $O(N)$ [1]. As an effect, the CCPT increases as the system grows, containing more and more nodes. The blockchain systems that use these algorithms are called scalable. For a small number of nodes this does indeed increase the performance greatly, but when popularity increases and a system runs over hundreds of machines the performance degrades [2]. This renders the system less usable for practical applications like global money transfers, where high throughput (2000 transactions/sec) is necessary, considering the high number of users [3].

The systems that do not have this limited transaction rate are said to *scale-out*. These systems have a CCPT of $O(1)$ and are therefore not affected by the number of nodes that are connected in the system. These schemes achieve this by compromising the termination property: transactions may only be known by a relevant subsection of all nodes. In practice, these schemes also have to compromise on the validity or agreement properties of the system [4]. This means that two honest nodes may disagree on the validity of a transaction, or that invalid transactions can be verified by honest nodes.

In a paper by Ren and Erkin [4] a solution is proposed that does not compromise on agreement and validity and is upper bounded by $O(N)$, making it scalable. In addition to this, it allows to reach scale-out throughput using *spontaneous sharding*. However, the system described in this paper is fully theoretical. In order to verify the claims that such a system scales out, we worked on an implementation of the described system. This technical report describes our work on the experimental implementation of the theoretical system from the paper. With this implementation, we aim to show the performance of the described algorithm, and ultimately show if it does scale out.

This report starts by a quick overview of the workings of the system described in the paper in chapter 2. After that, chapter 3 gives an overview of the system we designed that implements the blockchain. Lastly we show the results of the tests we conducted in order to verify if the system scales out in chapter 4. In the appendices we have included a more in depth explanation of the system components in Appendix A. Lastly, Appendix B shows the implementation specifics of algorithm 1 through 3 described in the paper.

[1] See the end of 2017/ begin of 2018 in the graph on `https://charts.bitcoin.com/chart/price`

2

Problem Statement

For the project, we had to implement the system described in the paper by Ren and Erkin [4] in order to verify if the described system does indeed scale out. A definition of our problem would therefore consist of a definition of the system we have to implement and what we need to show for proving this property. In this chapter we only give a simple overview of this system for brevity, and refer you to the original paper and the appendices for a more elaborate explanation and specific implementation details respectively.

2.1. Chains

The scheme described in the paper by Ren and Erkin [4] is an off-chain based blockchain. This means that in addition to a main chain, each nodes has his own local chain containing his own transactions. The node will commit an abstract of his local chain every once in a while to the main chain. The main chain then facilitates all the nodes reaching BFT consensus on the committed abstracts.

When a node sends a transaction it must also supply a proof of the validity of the transaction. The sender is incentivised to do so correctly, as this corresponds to proving that his money is real. A receiving node will verify the proof and will accept the transaction when the verification is successful and an abstract of the block containing the transaction has reached the main chain. The receiving node also stores this verified proof as it will be used to prove any future transaction that he makes with the received money.

2.2. Sharding

The transaction costs can be reduced by reducing the size of the attached proof. A rational node will therefore always try to minimize the size of the proofs. In order to achieve this, a node keeps track of what all other nodes know, and as an effect which parts of a proof are already known an can be pruned off. With this knowledge a node can select the unspent transaction with the smallest required proof to continue on. This behaviour spontaneously shards the network as certain transactions will always cycle in a small group to keep proof sizes as small as possible.

2.3. Scale-out

As already mentioned before, the communication cost per transaction is determined by the proof size. This makes the CCPT performance $O(p)$, where p is the size of a proof. In the worst case each proof requires chains of each other node in the system. This gives a worst case performance of $O(N)$ for the system. But as already mentioned, a rational node will try to reduce its proof sizes to the size of a small shard of size g. This makes that the CCPT can reduce to $O(g)$, which makes the system achieve scale-out throughput.

With our implementation, we aim to show that such a throughput can indeed be achieved. For this we need to observe the sharding behaviour when running the system, as this marks the crucial step in making a scalable scheme scale-out.

3

System Overview

In this chapter a high level overview of our system is given. For a detailed description of the working of the system we refer you to Appendix A and B, in which each component is explained in more detail. Our system consists of roughly three parts, the local system, the main chain and the tracker server. Each of these parts is described separately in the following sections.

3.1. Local system

The local system implements most of the functionalities described in the paper. Most importantly this part models all the nodes and their local chains. In order to allow for sending transactions to other nodes, this part of the system also includes node-to-node communication. Supplementing this communication is the code that creates transactions and validates them on receiving. These are the actual implementations of algorithm 1 through 3 as they are described in Appendix B and the original paper. Lastly, this part is linked to the main chain such that it can commit its abstracts and query for others.

This entire system is then wrapped inside an application and a simulation in a way that allows us to quickly set up a test with any number of nodes. See section A.6 for how this can be used.

3.2. Main Chain

The main chain, as described by Ren and Erkin [4], can be any blockchain that provides two functions. First, it must allow for a node to commit an abstract to the chain. Secondly it must have a way to check if a certain block is on the chain. Because it is only required that those two functionalities are provided, we opted to use an existing blockchain solution to cut down on the time it would take to develop our own implementation of a blockchain (including its consensus algorithms). The existing solution we decided on is Tendermint[1]. This blockchain uses PBFT for its consensus and has an API which can be accessed via most programming languages. As we decided to use Java for the main development, this API allowed us to easily integrate Tendermint. In addition to this, the original paper explicitly mentioned the use of PBFT for the main chain. Therefore we concluded that Tendermint would be a good implementation to use for the main chain. Even though we had a specific technology in mind, we created an interface `MainChain` in our code that exposes the two functions described above to the local system, so that any future change of the main chain technology is also possible. For Tendermint, we implemented this interface such that the internals are nicely hidden behind a layer, improving readability. For this implementation we made use of the jABCI[2] library.

3.3. Tracker

To coordinate all the nodes, a tracker server was implemented. It provides a way for nodes to register and get information on other nodes. This allows for distribution of public keys, IP-addresses, metrics to all the other nodes. The server is also used to determine if the simulation can start. The tracker is created in nodeJS[3] and provides endpoints/routes for nodes to communicate to via simple HTTP requests.

[1] `https://tendermint.com/`
[2] `https://github.com/jTendermint/jabci`
[3] `https://nodejs.org/en/`

4

Test Results

In order to show whether the system reaches scale-out throughput, we have conducted a number of tests. In these tests we used randomly generated transactions in uniform random time intervals to check whether the systems produces sharding behaviour. This sharding would enable our system to reach scale-out throughput.
We ran 4 different tests, with 10, 15, 20 and 25 nodes. The parameters that we used are shown in Table 4.1. The minimum time and maximum time between transactions had to be increased to prevent problems with Tendermint. However, this does not affect the transaction pattern, only the speed at which the simulation progresses.

Nodes	Min. time	Max. time	Commit every	Required commits
10	100	200	10	2
15	400	500	10	3
20	1200	1300	10	3
25	1200	1300	20	3

Table 4.1: The parameters of the 4 different tests.

The runs with 10 and 15 nodes were stable and were ended manually. The runs with 20 and 25 nodes were not stable and ended because of Tendermint failures. Any attempt to run with more than 25 nodes resulted in a failure within the first 2000 transactions, and as such we decided to leave those out. The results are presented in Figure 4.1.

Figure 4.1: A graph showing the average number of chains in each transaction

4.1. Sharding

Due to the small number of nodes, we are not able to definitively show sharding behavior. However, we can see that there seem to be asymptotic bounds on the average number of chains that are sent, which could indicate sharding behavior.

4.2. Notes and future work

During the testing of our system we found a number of things that could be improved. Due to time reasons we were not able to make all of these improvements. Therefore we chose to include this section as a starting point for further development.
First of all, we found that the disk IO of Tendermint forms the bottleneck of our system when running over 10 nodes, even on an SSD. One possible workaround for this issue is to use a ramdisk for the Tendermint nodes. This allowed us to run significantly more nodes on a single machine (25 instead of 10).
Another the problem we identified is that the Tendermint processes can get out of sync. Some process can be updated with all the block until height 5, while another is still waiting for block 3 to be finalized. As the gap between nodes grows, it becomes increasingly likely that a transaction is rejected because it cannot be verified by the receiver, even though the sender believes it can be. This can be fixed by sending the required height of the main chain along with the proof. A node can then check to see if they are up to date and otherwise wait until the required block is known in order to verify the proofs correctly.

A

Detailed system overview

In chapter 3 we introduced the workings of our system. This chapter goes into more detail on the specifics of that implementation. It does so by first introducing the data model we use in section A.1. In section A.3 the serialization scheme is discussed, followed by the communication between the nodes in where this serialization is used in section A.2. section A.4 and A.5 give insight in the working of the tracker and the used main chain respectively. Lastly a section is included that describes how the simulation of the system works.

A.1. Data Model

The main data model of our system is placed in the package `nl.tudelft.blockchain.scaleoutdistributedledger.model`. In this section an overview is given for the main components in this data model.

A.1.1. Node

Each node that is connected to the blockchain is modeled in our system as a `Node`. On each actual node a list is kept of all connected nodes to keep track of information we know about it. These objects therefore store the id, address and public key alongside a local chain and the meta knowledge (see subsection A.1.1). For nodes that run locally we also need to keep track of the private keys, therefore we extend `Node` with a private key in the class `OwnNode`.

Meta-knowledge We keep track of what each node knows. We call this `metaKnowlege`: what we know that they know. Each node contains a Map from nodes to integers, where the integer represents the last block that we know for sure that the receiver has.
The meta-knowledge of a node is updated whenever we receive a transaction from that node and when we send a transaction to that node.

A.1.2. Chain

This class represents the local chain in the algorithm. It contains exactly what you expect in that it has a genesis block, an owner and a list of blocks.

A.1.3. Main Chain

To streamline the interaction with the main chain we created an interface `MainChain` that defines a number of methods that allow for easy querying of the chain and committing blocks to it. The class actually implementing this interface is `TendermintMainChain`. For a in depth explanation of the workings of this class, see section A.5.

A.1.4. Block

A block can be one of two things, either it is a genesis block or it is a normal block. A genesis block is identified by its ID being zero and having no previous block, where a normal block has a positive ID and has

a link to the previous block (to form the blockchain). Besides this, a block also needs to contain transactions and we included some getters for certain properties (like 'is this block on the main chain?').

A.1.5. BlockAbstract

The `BlockAbstract` class contains all the fields that are required by the algorithm. These fields are the hash of the contained block, a block number, a signature and an owner. Additionally the class contains methods to check the signature, to convert itself to a byte array that can be committed to the main chain, and a conversion back when receiving from the main chain.

A.1.6. Transaction

To represent a transaction the `Transaction` class is used. This class contains no real surprises as it simply contains fields for the sender, receiver, sources, amount and remainder.

A.1.7. Proof

The `Proof` class is one of the more interesting classes mentioned in this section. This is because it includes code that reconstructs blocks and transactions when the proof is received from another node and the algorithm for proof verification as described in Appendix B.

Smart proof collection
When sending a proof for a transaction, we use our meta-knowledge of the receiver to select the blocks that we send. We send only the part of the chain that we expect that the receiver doesn't have, up to the last committed block of that chain. Combined with Algorithm 3, this ensures that we send as little information as possible.

A.2. Node Communication

Nodes have to communicate with each other about multiple things. The communication revolves mainly around sending and receiving transactions and proofs. For this communication, we make use of Netty [1] sockets in combination with custom messages. In order for each node to know where to send the messages to, each node knows the address of all other nodes via the tracker.
In addition to the two message types listed above, there are some other messages that the master machine sends to all nodes to control the simulation. First the `TransactionPatternMessage` distributes the transaction pattern that will be used during the simulation. Then the `Start-` and `StopTransactingMessage` will notify the nodes when to start and stop the simulation.

A.3. Serialization

When sending data to other nodes we cannot simply pass the references to the objects to the other nodes. This is because our system can run distributed and therefore it does not necessarily run inside a single JVM. As an effect we need to be able to serialize many objects in order to send them. Due to some aspects of our data structure, just simply serializing all the objects and deserializing them when receiving does not work. Therefore we have to perform some additional steps on the receiving side in order to fix the data structure. The only message passed between nodes that needs serialization is the `proof` . The proof contains pointers to transaction sources, blocks and owner of each of them. So to be able to deserialize it we need to replace the pointers by their corresponding identifiers (of type integer): identifier of the owner, number of transaction and number of block. On the deserialization part we just need to convert the identifiers back to pointers. To do this we use our knowledge and the chains from the proof.

A.4. Tracker Server

In some cases, nodes need a point of centralization (for example to bootstrap the simulation); for this case we use a tracker server. It can be setted up by installing NodeJS and running `npm install` in the `tracker-server` folder. After this it can be run by running `npm start` in that same folder.

[1] `https://netty.io/`

A.4.1. Routes

The tracker defines the following endpoints to connect to:

- `/`
  Get all nodes registered to the tracker. For each node its id, public key and address are sent.
- `/update-node`
  Updates a node with the given id to a new address and public key.
- `/register-node`
  Register a new node with a given id, address and public key.
- `/set-node-status`
  Update the running status of a node. This can be either `true` or `false`.
- `/node`
  Gets information of a note with the given id.
- `/reset`
  Reset the node list on the tracker server. This should only be called once when starting a simulation before any node registers, otherwise information about registered nodes is lost.
- `/status`
  Get the number of currently registered nodes and currently running nodes on the tracker server.
- `/demo`
  Show a live view of the demo. This demo view shows a live graph and counters for the total number of transactions and the average amount of blocks and chains sent per proof.

A.4.2. Simulation Coordination

When running a simulation, all nodes have to have certain knowledge about each other in order to coordinate the run. When starting the simulation all nodes have to generate their key-pairs and distribute the public keys to all other nodes. They also need to know the addresses of all the other nodes in order to communicate with them. The tracker facilitates this first step by making all the nodes register with their id, address and public key. After all nodes have registered, which can be checked by getting the status of the tracker, each node will request the list of registered nodes to get all the needed information. With this information the nodes will start their main chains and connect them to all other nodes. Before sending any transaction, the nodes should be sure that all other nodes are ready to receive them. In order to achieve this, a node will send an update to the tracker that it is ready when it is able to receive transactions from other nodes. The nodes will then, again via the `/status`-channel, wait until all nodes are ready before starting. For shutting down, a similar system is used that marks the nodes as stopped when they are done sending transactions. This way every node waits to shutdown until it is sure that no message will be received after it (up to a certain level due to it being an asynchronous system).

A.5. Main Chain

As already mentioned in chapter 3, we use Tendermint for our main chain. Tendermint exposes the ABCI interface, which allows us to commit to and read from the chain shared between the Tendermint nodes. This section elaborates on the implementation details of running Tendermint, integrating it with our own system and connecting different nodes together.

A.5.1. Running tendermint

Tendermint process can be spawned using the `TendermintHelper` class, which takes care of everything that is needed to start it up. The steps are usually:

1. Generate priv_validator.json file for each node.
2. Generate genesis block, which is then passed to calculate the initial application hash.

3. Generate genesis.json file for each node.
4. Run tendermint process for each of the nodes (this means there is a tendermint processes running at the same time, one for each node).

When running a simulation, these steps are all automatically taken by the simulation initialization. Alternatively, running Tendermint can also be done with the following command: (the use of which is discouraged though) `./tendermint node -consensus.create_empty_blocks=false` (Note that Tendermint creates empty blocks by default). For this method to work, you need to have a config.toml file, genesis.json and priv_validator.json for each node, and/or specify more parameters.

A.5.2. Connecting nodes

In order for the Tendermint process to know where to find other nodes you must present it with a list containing the addresses of the other nodes. This list is given as an argument when starting the process by appending the following argument `-p2p.seeds=` . Here the addresses are written as a comma separated list. According to the documentation there should be a way to dynamically add new peers to the system, but this functionality does not seem to be present in the current version. This is not a problem however, as we assumed a permissioned system in where all nodes are known at the start.

A.5.3. Communication over ABCI

The communication between the Tendermint process and the JVM, runs over the ABCI interface. This interface exposes a number of endpoints over HTTP, that allow you to communicate with the Tendermint core. Note that the endpoints listed in the Tendermint documentation are outdated, at the moment of writing, and that the actual endpoints can be found by opening `localhost:<TM_PORT>` with the executable running.
In our code this communication is wrapped in the `ACBIClient` class. This class contains the `sendRequest` method for sending the requests that returns a JSONObject of the response. Using this method, the other methods allow you to query the main chain for blocks at a specific height or with a specific hash and committing blocks onto the chain.

A.5.4. Caching

As sending a query to the main chain may be slow and is done many times in our system, in order to validate blocks, we implemented caching for the main chain to speed this up. This caching mechanism stores hashes of all received blocks when their containing block is committed. As an effect it can quickly be checked if a block is already on the main chain. When a hash is not found, it tries to update its cache and try again. It should also be mentioned that could take some time between committing a block o the main chain and it reaching the cache. It is therefore advised, and also implemented in such a way, that each node will only validate a block when there are more recent blocks already committed (making it reasonable to assume that al nodes are able to check that its on the main chain).

A.6. Simulation

For testing the system we added a way to easily create a simulation. In this simulation we can configure the number of nodes, change used transaction patterns and even run in a distributed way on multiple machines. In subsection A.6.1 the process of setting up the simulation is discussed and subsection A.6.2 shortly give an overview of the usage of transaction patterns.

A.6.1. Setup

When setting up the transaction you need a number of things before starting. First you need to decide which of the machines is the master that controls the simulation. Then you need to start the tracker on **one** of the machines and get the IP address of that machine. Lastly you have to decide on the number of nodes each machine should run and how many there will be in total. With this information you can start a simulation by applying the following steps on every machine:

- In the `SimulationMain` -class

  - Give each machine its sets of nodes by changing the `LOCAL_NODES_NUMBER`, `TOTAL_NODES_NUMBER`, `NODES_FROM_NUMBER` values
  - Specify the parameters for the transaction sending behaviour of each node using the fields `MAX_BLOCKS_PENDING`, `INITIAL_SENDING_DELAY`, `SENDING_WAIT_TIME` and `REQUIRED_COMMITS`.
  - Set `IS_MASTER` to `true` for the master machine and to `false` for all the others
  - If the current machine is the master, also specify the simulation time in seconds.

- In the `Application`-class

  - Set `TRACKER_SERVER_ADDRESS` and `TRACKER_SERVER_PORT` to point to the tracker location

With the configuration done you can run the simulation by calling the main method in the `SimulationMain`-class. Make sure to run the master machine first, as the master will reset the tracker on startup. For a live visualization of the network you can open `<tracker-address>/demo` in a browser.

A.6.2. Transaction Patterns

To allow for easy simulation setups, it is nice to have way way in which to describe the transaction sending behaviour of a node. For this reason we created transaction patterns that allow for exactly this. These patterns all implement the interface `ITransactionPattern`, which defines methods for making and sending transactions and committing them to the main chain when needed. Implementations of this interface therefore only have to implement the selection of the receiver and the amount that are being sent. Currently we have implemented three different patterns. First we have two random patterns that select a random amount and send this to a random other node. One of these uses a uniform distribution, while the other uses a more realistic Poisson-distribution. The third pattern is mainly used for debugging of the system as it will only allow node 0 to make transactions (These transactions are then uniformly distributed).

B

Algorithms

B.1. Algorithm 1: Transaction Validation

The implementation of this algorithm follows the pseudocode in the paper relatively closely. The main difference is that we use exceptions to clearly distinguish between the different failure causes. We have split the validation into three methods.

B.1.1. Check money

This method checks if the sum of the remaining values from the sources adds up to the amount and the remainder in the transaction. We use a fail-fast technique to quickly report failure whenever the sum becomes larger than expected.

B.1.2. Check double spending

This method checks that there is no double spending. It is equivalent to the pseudocode in the paper.

B.1.3. Validate sources

This method performs the source check. We use caching to prevent validating the same transactions multiple times. If a transaction has been validated before, we will use the cached result.

B.2. Algorithm 2: Proof Verification

The implementation of this algorithm also follows the pseudocode in the paper closely, with some minor optimizations. First of all, we do not check the block hashes and signatures when looking for committed blocks. This is because a blockabstract is only accepted on the mainchain if both the hash and signature are correct. Therefore checking if the blockabstract is on the mainchain is enough. We also fast-fail if a transaction is seen two times. Finally we cache if a transaction has been verified locally before, in that case we can quickly skip the verification of that transaction. This speeds up the algorithm especially when verifying transactions with a long chain of sources.

B.3. Algorithm 3: Non-interactive Smart Transacting Algorithm

The algorithm in the paper basically requires the use of a powerset of all unspent transactions. Since this requires $O(2^n)$ sets, this is only feasible for small numbers of n.
The algorithm implemented is smarter and eliminates bad combinations by keeping track of a best-so-far. It has the same worst case performance, $O(2^n)$, e.g. if all unspent transactions are required for the transaction. If this is not the case however (you send a part of your money), then the performance of the implemented algorithm is much better. The less transactions are required as sources, the better this algorithm performs. The algorithm also benefits from sharding, as transactions with the same requirements (e.g. received from the same source) will be considered as a group instead of individually.

B.3.1. Step 1
First we collect all unspent transactions (these are tracked in application). We then create a TransactionTuple for each transaction, which will calculate which chains would need to be sent based on what the receiver already knows. We then merge the tuples that have the same chain requirements in O(|transactions|) by hashing them to buckets.

B.3.2. Step 2
Check if there are single transactions / single groups that have a large enough amount to be used as the only source. If we find at least one of these, then all transactions that require more chains than the best single source are directly eliminated. (Combining those transactions with other transactions would only make the required set of chains bigger, so they will never be in the best choice)
We call the resulting set ‘candidates‘.
If there are less than two candidates we stop and return the best-so-far.

B.3.3. Step 3
We have a set of TransactionTuples ‘currentRound‘, which is initially set to ‘candidates‘. We have a set of TransactionTuples ‘nextRound‘, which is initially empty.
For at most `candidates.size() - 1` rounds, we do the following:

- We iterate over the candidates. We try to combine each candidate tuple with each element of ‘currentRound‘.
  - If the combination requires more than or the same number of chains than the best-so-far, then it is eliminated.
  - Otherwise:
    - If it is able to cover the costs, we set it as best-so-far.
    - Otherwise, we add this combination to ‘nextRound‘.
- If there are less than two combinations selected for the next round, then the algorithm returns the best-so-far.
- Otherwise, `currentRound := nextRound‘ and ‘nextRound := []` .

B.3.4. Result
The final combination of transactions returned is the set of sources with minimum amount of chains required. The set found is the largest possible set for its chain requirements, but not necessarily the largest possible set.

B.3.5. Performance
The performance of the algorithm is still $O(2^n)$ in the worst case. The worst case would be if all transactions are needed to get the amount of money required and none of them can be grouped. Then the algorithm will consider all possible combinations.

B.3.6. Future optimizations
- It would probably be useful to keep track of how much money we have, to prevent the costly case where we don't have enough money.
- There could be 2 algorithms, one for when we expect many transactions will be required, one for when we expect only a few transactions will be required. The current algorithm works bottom up, the other algorithm could work top down. (How many transactions do we need at minimum to cover the transaction?)

B.3.7. Implementation Details
BitSets are used as a convenient and efficient way to represent the chains that are required. They represent a sequence of bits of a particular size (e.g. 1000 bits).

- The bit at index 0 represents chain 0, the bit at index 1 represents chain 1, etc.

- A bit with a value of 1 means that the corresponding chain is required.
- Combining the chains required of 2 tuples is done with a bitwise or.
- The number of chains required is the cardinality of a bitset.
- Removing the chains that the receiver already knows is done with a bitwise andNot.

Bibliography

[1] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[2] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[3] Visa Inc. Visa inc. at a glance - june 2015. Visa, 2015. URL `https://usa.visa.com/dam/VCOM/download/corporate/media/visa-fact-sheet-Jun2015.pdf`.

[4] Zhijie Ren and Zekeriya Erkin. A scale-out blockchain for value transfer with spontaneous sharding. *arXiv preprint arXiv:1801.02531*, 2018.