University of Applied Sciences
Rapperswil

Term Project

Scalability for the Bazo Blockchain with Sharding

Author:
Roman Blum

Supervisor:
Dr. Thomas Bocek

A project submitted in fulfillment of the requirements for the degree of Master of Science in Engineering

in

Information and Communication Technologies

December 12, 2018

Declaration of Authorship

I, Roman Blum, declare that this thesis titled, "Scalability for the Bazo Blockchain with Sharding" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"*What does Minecraft and the future I envision have in common? – They're both built on blocks.*"

Roman Blum

UNIVERSITY OF APPLIED SCIENCES RAPPERSWIL

Abstract

Master of Science in Engineering

Scalability for the Bazo Blockchain with Sharding

by Roman Blum

The blockchain trilemma claims that distributed ledger systems are subject to an impossible trinity between decentralization, scalability and security. Such systems can only achieve two out of these three properties to a highest degree. However, with the increasing popularity of cryptocurrencies and smart contract platforms, all three properties are required to its full extent in order to facilitate future adoption.

In this work, we present a sharding concept for the Bazo blockchain designed to scale as the number of users increases. With new users joining and using the network, the protocol load-balances the computational work by dynamically adjusting the number of partitions (shards). The core of Bazo's consensus protocol relies on *self-contained proofs*, a new primitive for blockchain protocols to prove that a user has sufficient funds without a validator relying on the full blockchain history.

We further propose a mechanism to reduce the overall size of the blockchain over time. The mechanism follows a *transaction aggregation* style using coalesced transactions and collectively signed state blocks. Within the realm of this new concept, we provide examples of in-shard and cross-shard transactions and briefly discuss mitigated attack scenarios to preserve the consistency and integrity of the Bazo blockchain.

Acknowledgements

I sincerely want to thank my advisor and colleague Dr. Thomas Bocek who provided insight and expertise that greatly assisted the research. The weekly discussions were of great help and him sharing his pearls of wisdom were always helpful and appreciated.

Contents

Chapter 1

Introduction

Blockchain brings a paradigm shift to many businesses and the possible areas of application are still unfathomable at present time. Financial institutions, governments, healthcare businesses, supply chain firms, among others, are starting to realize that blockchain is not an exuberant idea dreamt by cyber-libertarians praying for utopian ideals. It has become increasingly evident that decentralized networks, distributed ledgers, and token economics could propel industries into a new era of enterprise.

Nevertheless, while cryptocurrencies progressively enjoy mainstream adoption and the interest in blockchain technology grows on a daily basis, major challenges remain. There is still one missing piece that prevents these applications come to fruition on a global extent: *scalability*.

1.1 Motivation

The impossible trinity between decentralization, scalability and security [10] that blockchains suffer to can be explained with Bitcoin as an example. The prominent cryptocurrency with the highest market capitalization, around $110 billions at the time of writing [7], roughly processes 4 transactions per second (tps) on average. In contrast, Visa has a peak capacity of 65'000 tps [13].

Bitcoin's challenging limitation lies in the fact that each full node is required to store the complete blockchain, which is a continuously-growing list of transactions, and by now takes up around 160 GB of hard drive space. The number of transactions per second is constrained by the maximum block size and the block confirmation time. Additionally, each full node has to perform all computations and validate every single transaction performed on the network. As a result, the number of transactions the blockchain is able to process can never exceed that of a single node.

A viable, but in some cases naive approach to increase the transactional throughput would be to simply increase the block size limit, as stated in several proposals [2, 11]. However, the past has shown that increasing the block size limit is a highly controversial topic [4] and it has been a dividing factor of the Bitcoin community, notably the hard fork of Bitcoin and Bitcoin Cash [17]. In the case of Bitcoin and Bitcoin Cash, the transaction throughput indeed increased after the fork due to the higher block size limit, but since the community split in two, the computational resources, hence the overall security of each blockchain proportionally dispersed.

In traditional environments, scalability is solved by adding more servers to process more transactions, or to generally handle an increasing demand of incoming network traffic. In contrast, scalability in a decentralized environment *can be solved* by adding more computing power to every node for the network to get faster. This solution could postpone the effects of network limitation, but it's likely to only be a temporary solution if Moore's Law [21] is in effect for these resources.

In general, raising the block size limit makes running a full node more difficult, since the required resources to participate in the network increases. The more computationally expensive it is to validate the blockchain, the fewer network participants who will do it. This leads to a risk of much higher centralization due to the tragedy of the commons [12].

After all, the increasing popularity of cryptocurrencies and smart contract platforms shows that the necessity for a future-proof scalability solution is an inevitable challenge for Bitcoin, Ethereum and every other blockchain-based consensus protocol.

1.2 Description of Work

This thesis covers the design of a sharding mechanism for the Bazo cryptocurrency that allows the blockchain to scale and greatly increase the transaction throughput. Bazo [3, 24] is an open source cryptocurrency developed from scratch by students at the University of Zurich and University of Applied Sciences Rapperswil. Initially developed as a blockchain based on Proof of Work (PoW), energy consumption considerations and the risk of centralization were reasons to recently switch from PoW to Proof of Stake (PoS).

In PoS, validators vote with their ownership of a certain cryptocurrency unit instead of computational power. The weight of each validator in the network depends on the size of of their stake. PoS seeks to address the computational overhead of PoW by attributing mining power in proportion to the amount of coins held by a miner

The importance of finding a solution to the scalability problem has raised awareness in the academia and industry [9, 16, 26]. Blockchain scaling is an on-going research in this area and the viability of blockchain sharding must still be established. Thus, this conceptual work is highly explorative and prototype-driven.

1.3 Project Outline

The remainder of this document is structured as follows: Chapter 2 elaborates on three different scaling solutions for blockchains and familiarizes the reader with sharding. Chapter 3 introduces a new design approach for sharding blockchains. Chapter 4 gives further implementation details and elaborates on practical details. Chapter 5 conducts an evaluation of the chosen sharding design. Conclusions are drawn in Chapter 6.

Chapter 2

Background and Related Work

This chapter presents three scalability solutions and specifically familiarizes the reader with sharding, a scaling solution with the aim of improving transaction throughput and reducing per-node resource requirements.

2.1 Solutions

Building a scalable blockchain has been under scrutiny for quite a while and compelling efforts have been put into research and development. The fundamental problem that prevents blockchain to scale is the underlying protocol. The consensus architectures strongly rely on every node processing every transaction because the security of the network relies on the percentage of honest users. Fortunately, a number of ways to solve the non-scalability of blockchains arose in the past and have produced several proposals from both academia and industry.

This section covers the most promising solutions, namely state channels, Plasma and sharding. It is important to point out that all of them are complementary in the sense that each solution has their advantages and disadvantages over the others and maximum scalability could be achieved when all three solutions are in use in parallel. Furthermore, another key point of solving scalability of blockchains is the introduction of PoS [25], because it is much easier and faster to reach consensus by checking the stake of a node than it is to check the hashing power. However, PoS will not be discussed further in this paper.

2.1.1 State Channels

A state channel is a two-way communication channel which allows participants to send transactions off-chain, that is, participants do not rely on the state-altering operation of the blockchain, but instead can rapidly perform an unlimited number of updates without the need to involve the blockchain at all.

State channel solutions are currently being developed and tested for both Bitcoin and Ethereum with the Lightning Network [23] and the Raiden Network [6] respectively. Without loss of generality, the following example explains state channels based on the Raiden Network specification.

The Raiden Network is an off-chain transfer network for ERC20 tokens. While Ethereum smart contracts enable secure on-chain settlement, Raiden enables off-chain payment transfers. It is simply described as follows: Assume Alice and Bob want to transfer assets with each other. To open up a channel, Alice simply deploys a smart contract on the Ethereum blockchain. Then, both

parties make a security deposit to the channel, i.e., they both deposit 5 Ether and the smart contract holds these assets on their behalf. Once the channel is open, they can digitally sign as many transactions between them as they like off-chain (up to 10 Ether) without ever broadcasting the changes to the network. In order to use the transferred assets on-chain, the channel needs to be closed either by Alice or Bob. The only information that is written to the blockchain is the net outcome of all transfers. The information about individual transactions stays private.

Raiden addresses a number of blockchain related issues:

Scalability Ethereum is currently limited to around 10 transactions per second. Raiden will scale with the number of its users, i.e., as the network expands, the maximum throughput will increase linearly.

Latency Ethereum mines a new block approximately every 15 to 30 seconds and finality of a transaction takes a few minutes. With Raiden, transactions immediately show up the moment they are sent. There is no need to wait for any confirmations.

Transaction Fees Opposed to paying fees for global consensus on the Ethereum blockchain, state channel participants only have to pay for forwarding peer-to-peer consensus which results in tiny transaction fees. In fact, transfers made via payment channels, unlike on-chain transfers, inherently do not require any fees.

Privacy When a state channel between two participants is settled, only the sum of transactions appears on the blockchain. Intermediate payment transfers do not appear on the blockchain.

If implemented correctly, Raiden can solve substantial issues of the Ethereum blockchain and ultimately brings low-fee, highly scalable, near-instant and privacy-preserving payments to the masses.

2.1.2 Plasma

Plasma [22] is a framework proposed by Lightning Network creator Joseph Poon and Ethereum creator Vitalik Buterin. Plasma is a layer two scalability solution which doesn't improve the blockchain itself. Instead, it takes an existing blockchain, creates a special construction which is connected to it, and thus provides a much higher throughput.

In essence, Plasma connects parent and child chains to the main blockchain and transforms it into a tree-structure. By using MapReduce functions and a combination of PoS token bonding, fraud proofs and chain exit strategies, Plasma can increase transaction throughput exponentially without sacrificing security that usually comes with smaller chains.

In Plasma, a series of contracts run on top of a blockchain, e.g. Ethereum. The blockchain acts as the root chain and allows additional chains to register themselves by a smart contract. Each parent chain, or "Plasma chain", has its own independent data and only periodically reports back to the main chain. One can view the main chain as the Supreme Court from which all subordinate courts

derive their power [1]. In order to detect and penalize fraud, parties of a parent chain are responsible for monitoring the particular chain they are interested in. In the event of an attack, participants can rapidly and cheaply do a mass-exit from the child chain to a root chain.

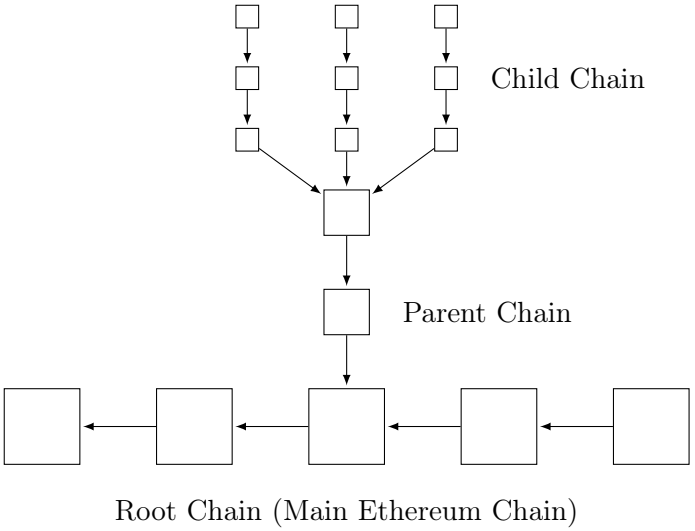


FIGURE 2.1: Plasma transforms the blockchain into a tree-structure using MapReduce functions that are run on Plasma chain trees.

Figure 2.1 shows that Plasma opts for a tree structure where the main blockchain is the root and the parent and child chains are the branches allowing for exponential increase in scalability.

2.1.3 Sharding

Sharding is a pattern [19] in distributed software systems and commonly utilized in distributed databases such as MongoDB [20], Elasticsearch [8], and Google's Spanner [14]. In these database systems, a shard is a horizontal partition, where each shard is stored on a separate database server instance holding its own distinct subset of data. Each shard typically contains items that fall within a specified range determined by one or more attributes of the data.

For example, a possible strategy is hashed sharding, where a hashed index of a single field serves as the *shard key* to partition data across clusters, as shown in Figure 2.2. The chosen hashing function distributes data evenly across the shards and reduces the load each shard has to deal with, albeit the additional overhead of computing the hash.

However, these sharding protocols do not take into account a Byzantine setting, where nodes must be resilient against malicious attackers, and in addition, independent nodes have to reach consensus over the current state of the network. In fact, sharding in a permission-less, decentralized network with the presence of Byzantine adversary is a well-known problem [18] with many challenges.

Sharding is a layer one scalability solution which is designed to directly improve the existing blockchain. In order to scale a blockchain with sharding, the state must be divided into partitions (shards), where each shard is stored on

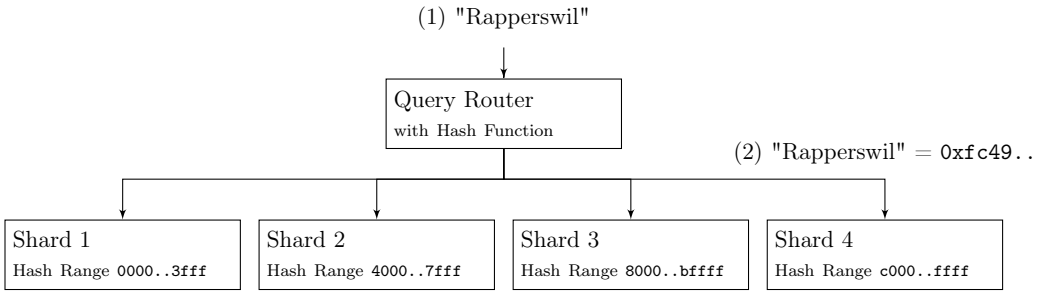


FIGURE 2.2: Hashed sharding is common in traditional databases where values are partitioned into shards by its shard key.

different nodes in the network. Nodes work on individual shards side-by-side, processing only a small part of the network state while still maintaining cross-shard consensus to prevent double-spending and other attack vectors. The point of sharding in a blockchain setting is to create parallelization, i.e., instead of 10'000 nodes processing, validating, and storing all transactions that happen on the network, 100 nodes process 100 transactions.

2.2 Challenges

Ultimately, our goal is to avoid requiring nodes to store the state of the entire system. We want the blockchain to scale linearly with every new node added to the network, also called *horizontal scaling* in traditional environments. We achieve this by dividing the network state into partitions, called *shards*, where every individual shard is responsible only for a subset of users and each shard having it's own blockchain. We want to achieve higher scalability without sacrificing security or decentralization. The protection of each individual shard must be ensured *implicitly* by the strength of a large fraction of the network, but without each shard having to watch other shards *explicitly*.

Chapter 3

Design

A detailed description of the sharding protocol design with Bazo as the underlying blockchain is covered in this chapter.

3.1 System Setting and Assumptions

This section outlines the system setting provided by the current state of the Bazo blockchain and makes assumptions about the future state that is desired at the end.

3.1.1 Entities

Bazo consists of four entities: *users*, *validators*, *leaders* and *aggregators*. A *user* is an entity who uses Bazo's infrastructure to transfer funds or run smart contracts. A *validator* is a node in the network who participates in Bazo's consensus protocol and validates blocks. While users only store block headers, validators store block headers and bodies of their assigned shard. For the sake of simplicity, examples in this paper assume that every user is a validator.

Furthermore, there are two special types of validators: A *leader* is a validator who has the right to append the next block to the blockchain of one particular shard. An *aggregator* is a validator who is responsible for the aggregation of transactions to reduce the overall size of the blockchain.

Each entity of the network has a public-private key-pair to digitally sign transactions. Bazo employs elliptic curve digital signature algorithm (ECDSA) as the base signing algorithm. In the rest of this paper, we refer to this ECDSA key-pair as to the user's *wallet keys* (pk_{wall}, sk_{wall}).

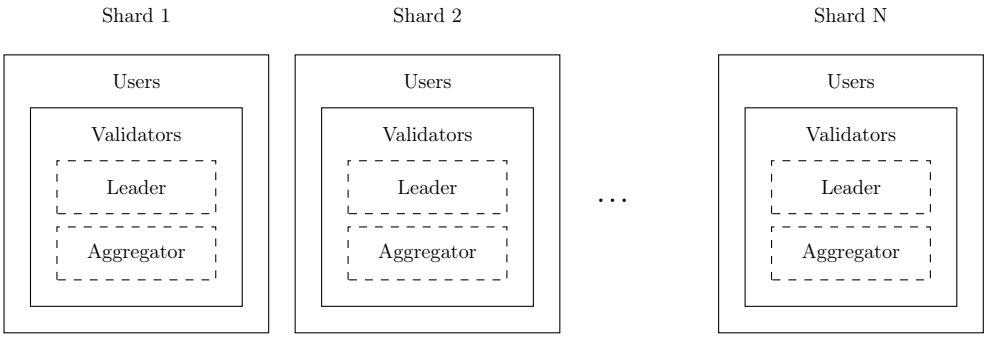


FIGURE 3.1: An overview of entities in the Bazo blockchain.

Figure 3.1 shows that entities are equally partitioned into shards 1 to N. While users and validators always remain in their particular shard based on their wallet address pk_{wall}, leaders and aggregators change block-wise. An aggregator is randomly elected in-shard and a validator is randomly elected cross-shard.

Furthermore, when a user wants to join the set of validators it must create an additional public-private key-pair using RSA and publish a stake transaction containing the public key. Note that this key-pair differs to the key-pair used to sign transactions. In the rest of this paper, we refer to this RSA key-pair as to the user's *commitment keys* (pk_{comm}, sk_{comm}).

3.1.2 Network Sharding

The Bazo infrastructure is divided into several subgroups referred to as *shard*, each having it's own blockchain called *shardchain*. Transactions of users are stored in a transaction pool by every validator in the network, but are only validated by validators based on the public wallet address pk_{wall} of sender or receiver. For example, a validator V of shard S is responsible for user U and validates transactions where U is either sender or receiver of funds in a transaction. Thus, user U is *assigned* to shard S.

Likewise, validators are only responsible for the maintenance of the shardchain based on their pk_{wall}. Maintenance is hereby defined as validating blocks and storing the full shardchain. Responsibilities of leaders are outlined in Section 3.3.

Note that users also receive blocks but unlike validators, they do not participate in the validation process, which is further explained in Section 3.3.2. Users in Bazo can be compared to SPV clients in Bitcoin, i.e., they only store their own transactions and block headers of the shardchain they belong to. To put it differently, validators store every block header received from the network, but only validate blocks based on their pk_{wall}.

3.1.3 Epochs

The algorithm proceeds in *epochs*. An epoch e ends after a predefined number of blocks, that is, assume that epoch length $length_{ep} = 100$. Epoch e starts at block number 1 and ends after block number 100, and the next epoch $e+1$ starts at block number 101 and ends after block number 200, and so on. Furthermore, the process of ending an epoch is called *finalization*. An epoch finalizes with a special type of block called *epoch block*, denoted *eb*. An epoch block serves as a marker. Epoch finality is further explained in Section 3.4.

An example is shown in Figure 3.2. The Bazo blockchain starts with epoch block eb^h and partitions the state into three shardchains sb^h_{1-3}, where sb_{1-3} is a shard block of shardchain between one and three, and where h is the block height. In this example, epoch length $length_{ep} = 3$. Thus, if three blocks have been created since the last epoch block, the current epoch is being finalized. Epoch block eb^5 is non-interactively created.

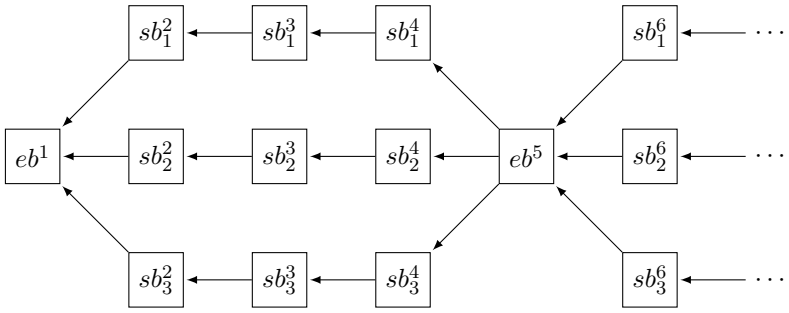


FIGURE 3.2: The concept of epochs with number of shards $NofShards = 3$ and epoch length $length_{ep} = 3$.

3.1.4 Number of Shards

The number of shards is an important property that determines the overall scalability of the blockchain. As the number of transaction increases, the computational resources and network bandwidth required for processing transactions increases for each individual node. A balance between the number of users and the number of shards is crucial because

1. having only one shard is equal to a blockchain without sharding,
2. having too few shards only scales sub-linearly as the number of users increases, or
3. having too many shards increases the communication overhead required to synchronize the block height.

Hence, a load-balancing algorithm is required to dynamically adjust the number of shards as new users are joining and using the network.

As a first approach, the number of shards could be determined by the sum of all staked coins of the network divided by a predefined constant value that specifies the staking power per shard. This approach has the advantage that the number of shards increases (or decreases) in proportion to the actual staking power of the network. Unfortunately, the number of shards could easily be faked by a single user with a large amount of staked coins.

A second approach would be to determine the number of shards by dividing the number of all validators of the network by a predefined constant value that specifies the number of validators per shard. This approach has the advantage that the number of shards increases (or decreases) in proportion to the actual number of validators of the network. However, the disadvantage of this approach is that the number of shards can easily be faked by a user with many wallets containing a few coins.

The last approach is to determine the number of shards by the Equation

$$NofShards = \left\lceil \frac{NofUsers}{NofUsersPerShard} \right\rceil, \tag{3.1}$$

where *NofUsers* is the total amount users of the network, and where *NofUsersPerShard* is a predefined value that specifies the number of users per shard. The

second step is to verify if the number of shards is lower than the number of validators, i.e.,

$$NofShards \geq NofValidators, \tag{3.2}$$

where *NofValidators* is the total amount of validators of the network. If *NofValidators* is lower than *NofShards*, Equation 3.1 is recalculated with a lower *NofUsersPerShard* until Equation 3.2 is true.

While the number of validators is known to each validator, it is important to point out that the total amount of all users is not known and cannot be easily calculated since every shard maintains a different amount of users. Instead, the number of users a shard maintains is contained in the header of every shard block *sb*, and a validator derives the number of users *NofUsers* by adding up the *NofUsersInShard* property of each block, that is,

$$NofUsers = \Sigma_{i=1}^{NofShards} UsersInShard(sb_i^h), \tag{3.3}$$

where h is the block height, where $1 \leq i \leq NofShards$ is the shard identifier, where *UsersInShard* is a function that returns the value of the number of users in shard i of a block. Note that validators only store blocks of their assigned shard until an epoch ends because the increase or decrease of the number of shards is unknown ahead of the next epoch resulting in a more or less partitioning of the network. For example, a validator can be responsible for blocks of shard 2 (of 3 in total) in epoch e and responsible for blocks of shard 3 (of 4 in total) in epoch $e + 1$.

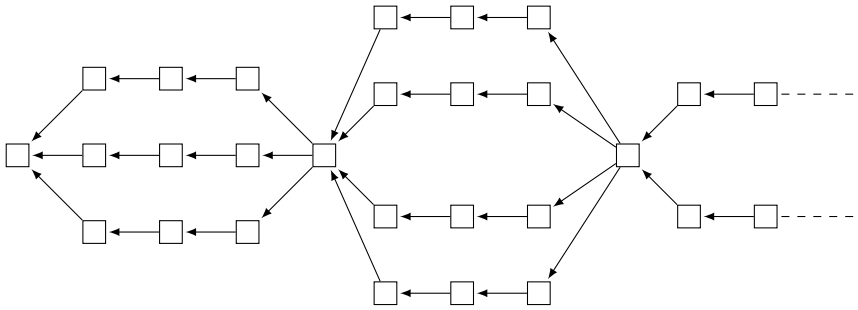


FIGURE 3.3: Load-balancing dynamically adjusts the number of shards to maximize throughput and minimize overload of a single node.

3.2 Identity Setup and Leader Formation

Validators are users that have joined the set of validators by publishing a StakeTx containing the public commitment key pk_{comm} to the network, which is explained in more detail in [5]. However, and in order for validators to get the right to append a block to a shardchain, they have to be elected as a leader and assigned to a shard.

3.2.1 Leader Election

The leader election process is identical to the process of fulfilling PoS condition introduced in [3], which was further revised in [5] due to a security vulnerability.

Each validator participates in the leader election process in a non-interactive way. Contrary to PoW, a validator is limited to exactly 1 H/s (hash per second) by providing a valid *TimeInSeconds* to fulfill the PoS condition

$$\frac{\text{SHA3-512}([Proof_{PrevBlocks}] \cdot Proof_{Local} \cdot Role \cdot TimeInSeconds)}{Coins} \leq Target, \tag{3.4}$$

where

$$Proof_{Local} = RSA(sk_{comm}, \text{SHA3-512}(BlockHeight)). \tag{3.5}$$

The PoS condition consists of the following properties:

List of the Previous Proofs ($Proof_{PrevBlocks}$) By including a list of the previous proofs of a particular shard, a stake grinding attack becomes infeasible.

Local Proof ($Proof_{Local}$) All parameters are the same for each validator in the network except the local proof. The local proof individualizes the PoS condition to each validator and therefore, a validator with a low amount of coins also has the possibility to append a block to the blockchain.

Election Role($Role$) Determines the role for which a validator wants to be elected for. This property is set to "L" for the leader election process and set to "A" for the aggregator election process.

Amount of Coins($Coins$) The amount of coins a leader owns. Note that the chances increase proportionally to the number of $Coins$.

Difficulty of the PoS condition($Target$) A validator has to fall below a *Target* in order to append the next block to the blockchain. Note that this target determines the speed of the blockchain. As the number of validators in-/decreases the difficulty must be adjusted accordingly.

If a validator falls below the $Target$ value, he or she becomes a leader. A leader has the right to append the next block to a particular shardchain. The system detects malicious attackers who attempt to speed up his hashing power, e.g. by manipulating the clock speed. The probability of a validator being elected to append the next block is in proportion to the stake he or she has. For example, a validator with 50 coins has a 5 times higher chance than a validator with 10 coins. The parameters in Equation 3.4 correspond to a validator's pk_{wall}, i.e., the validator uses the proofs $[Proof_{PrevBlocks}]$ and the block height $BlockHeight$ of the shardchain they are assigned to.

3.2.2 Leader Assignment

Permitting a leader to choose the shard he or she wants to append a block is insecure. Thus, we need a way to introduce randomness to securely assign a leader to a shard. True randomness is crucial to achieve a fair election among stakeholders, otherwise the algorithm may be prone to manipulation by an adversary [15]. Thus, we decided to assign a leader to a shard in a non-interactive way using only information that is locally available to each validator. A leader is assigned to a shard using parts of the PoS condition 3.4, that is,

$$\text{SHA3-512}([Proof_{PrevBlocks}] \cdot Proof_{Local} \cdot Role \cdot TimeInSeconds) \ mod \ NofShards, \tag{3.6}$$

Notice that part of Equation 3.6 is just the dividend of the fraction of Equation 3.4, and thus, the description of properties are identical, except the property *NofShards*.

Number of Shards ($NofShards$) A leader is assigned to a shard by using the remainder of the modulo operation with the number of shards the network is partitioned in to.

It is important to realise that this mechanism cannot ensure that every shard has a leader for a specific block height. At block height h, a shard can have zero or more (up to the number of validators) leaders. In the case that two or more leaders are assigned to the same shard, the leader with the lowest *Target* has the right to create the next block.

3.2.3 New Validators Joining Bazo

Users can express their intention to join the set of validators by sending a StakeTx to the network at any time. Since users only store block headers of the shard they are assigned to, a bootstrapping process to download the shardchains history would be required. The bootstrap time includes the time to download the block headers of a shard and the time to process the history necessary to validate the current system state. Hence, and for the sake of simplicity, new validators do not immediately join the set of validators and instead have to wait until the current epoch ends before they can participate in the leader election process.

3.3 Transaction Sharding and Processing

In Section 3.1.2, we presented a way on how to partition the network into shards, with each shard capable of parallelized transaction processing. In this Section, we present how transactions are assigned to shards.

3.3.1 Self-Contained Proofs

The core of Bazo's consensus protocol relies on *self-contained proofs* (SCPs), a new primitive for blockchain protocols to prove that a user has sufficient funds without relying on the blockchain history at all. Instead of validators requiring

the full blockchain to validate a transaction, a user has to provide all required proofs in the transaction in order for validators to verify a transaction, independent of the blockchain.

A self-contained proof consists of Merkle proofs provided by the sender of a transaction. A sender has to provide one Merkle proof for each transaction sent since the last epoch. Validators could know the number of proofs a sender has to provide by comparing the current transaction counter of an account with the transaction counter of the last epoch. However, performing checks against the transaction counter is not sufficient and further checks must be performed, elaborated in Section 3.3.4 and reasoned in Section 5.2.

Example 3.1. Consider a shardchain as shown in Figure 3.4. Assume that user A is also a validator and hence has the complete history of the shardchain since the last epoch. Further assume that user A has already transmitted two funds transactions $F3$ and $F6$. User A wants to create a new transaction and send it to the network. Therefore, a self-contained proof must contain two Merkle proofs, one for each transaction $F3$ and $F6$, i.e., $SCP = \{Proof_1, Proof_2\}$, where $Proof_1 = \{F5, F6, B2\}$ and $Proof_2 = \{F3, F4, A1\}$. Note that a SCP contains a Merkle proof for the most recent transaction first.

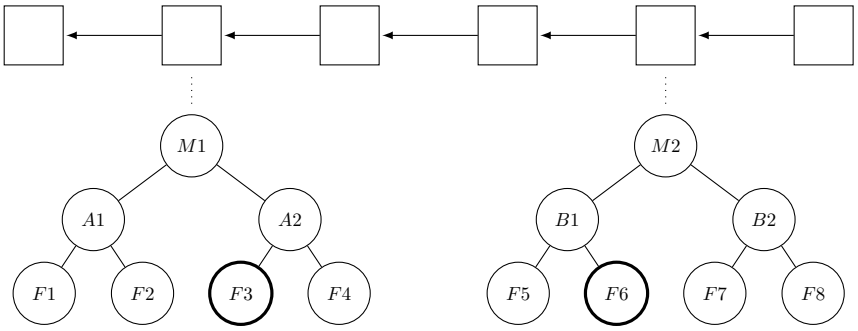


FIGURE 3.4: Initial situation of an SCP example: A user has to include two Merkle proofs for $F3$ and $F6$ in the next transaction.

This example also applies for transactions where user A received funds or aggregated transactions. In fact, performing block-wise transaction aggregation by sender performed by the leader creating a block helps mitigating fraudulent proofs, as evaluated in Section 5.2.

From example 3.1 we can also conclude a maximum size of a self-contained proof. Assuming that $length_{ep}$ is the epoch length and a shard block can contain up to m transaction of size s, we derive the upper bound of a self-contained proof being equal to

$$length_{ep} \cdot log(m) \cdot s = \mathcal{O}(length_{ep}). \tag{3.7}$$

3.3.2 Transaction Validation

Once leaders have been assigned to a shard, each leader has to create a new block with transactions based on the assigned shard and the transaction. Each transaction consists of a *self-contained proof* that refers back to the final state

of the last epoch. With self-contained proofs, validators are able to verify transactions without relying on external resources, i.e., only the final state of the last epoch and the self-contained proof contained in a transaction is required for verification.

Each validator locally maintains a transaction pool with all transactions of the network, and when a validator becomes a leader of shard, the following steps are taken by the leader at block height h to create block b:

1. The leader picks transactions from the local transaction pool with

$$s = pk_{wall} \ mod \ NofShards, \tag{3.8}$$

   where s is the assigned shard, where pk_{wall} is the wallet address of the sender or receiver, where *mod* denotes the mathematical modulo operation, and where *NofShards* refers to the number of shards the network is partitioned in to.

2. For each transaction, the leader verifies the signature of the transaction using the public key pk_{wall} of the transaction sender tx_{send}. If the signature is valid, continue. Otherwise, the transaction contains no valid signature and the algorithm stops.

3. The leader then proceeds to check the self-contained proof containing one or more Merkle proofs. Using the block headers of shard s, the leader performs the following subtask, iteratively going back starting with block sb_s^x, where $x \leftarrow h - 1$:

   (a) By querying the Bloom filter, the leader can check if sb_s^x contains a transaction of tx_{send}. If yes, continue. Otherwise the block contains no transaction of tx_{send}, set $x \leftarrow x - 1$ and repeat step.

   (b) Generate Merkle root r by using the Merkle proof provided in the self-contained proof. If $r = MerkleRoot(sb_s^x)$, where *MerkleRoot* is a function which returns the Merkle root of a block, set $x \leftarrow x - 1$ and repeat subtask for each Merkle proof until all proofs are checked. Otherwise, the self-contained proof is missing a Merkle proof not provided by tx_{send} and thus, the transaction is invalid and discarded. Section 5.2 elaborates on the necessity of this subtask.

4. Once a block is full, i.e., the maximum amount of transactions that fit into a block is reached, the leader adds a staking proof to the block. The staking proof proves that the leader has sufficient funds and that he or she fulfills the minimum staking amount requirement. The staking proof is also a self-contained proof containing one or more Merkle proofs.

5. Lastly, the leader follows the standard block creation procedure, which includes generating the Merkle root, creating the Bloom filter, hashing the block, etc.

Once this process is finished, the block is sent to the network for distribution.

Upon receiving block b, validators of shard s query for all transactions that are contained within the block and verify the validity. If b is valid, the block is

appended to the chain of shard s. Otherwise, the leader who proposed block b gets slashed and validators wait for another block for block height h.

Note that Bloom filters have a false-positive rate of 10% in our system. A user constructing a self-contained proof has to provide an additional proof for each invalid return value of the Bloom filter, i.e., the user has to prove that a transaction is *not* in the block that returns a false-positive. Since transactions (leaves) in a Merkle tree are in ascending order, a user only has to provide a Merkle proof that shows that the transaction in question is not part of the Merkle tree. Example 3.2 conceptually describes a possible scenario.

Example 3.2. Consider a block containing transactions $\{0x12, 0x15, 0x17, 0x18\}$. The Bloom filter of the block returns a false-positive for transaction $0x14$. Figure 3.5 shows the Merkle tree of the block with the leaves in in ascending order. In that case, a user has to provide a Merkle proof that contains $\{0x12, 0x15, H2\}$ deeming the false-positive invalid.

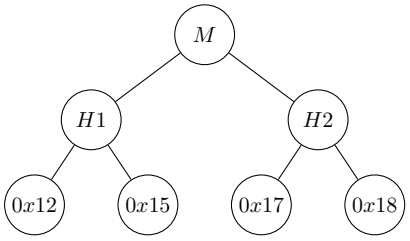


FIGURE 3.5: Transactions in Merkle trees are in ascending order

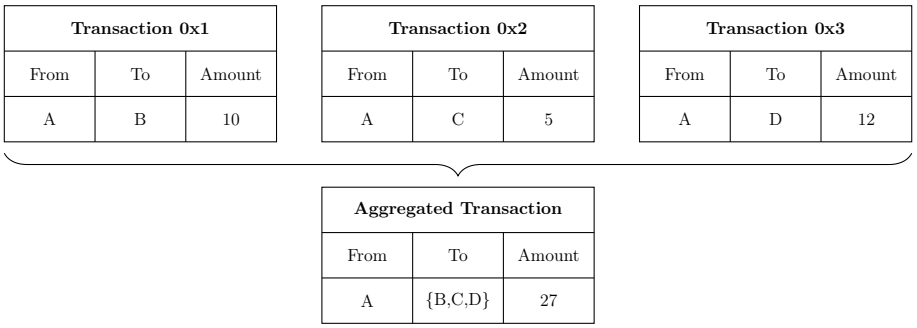
3.3.3 Transaction Aggregation

Transaction aggregation is a way of keeping the size of the ledger small, that is, validators only store transactions not older than a predefined number of blocks, denoted as aggregation length $length_{ag}$. A special type of transaction called *Aggregation Transaction (AggTx)* is created by summing up transferred amounts either by sender or by receiver.

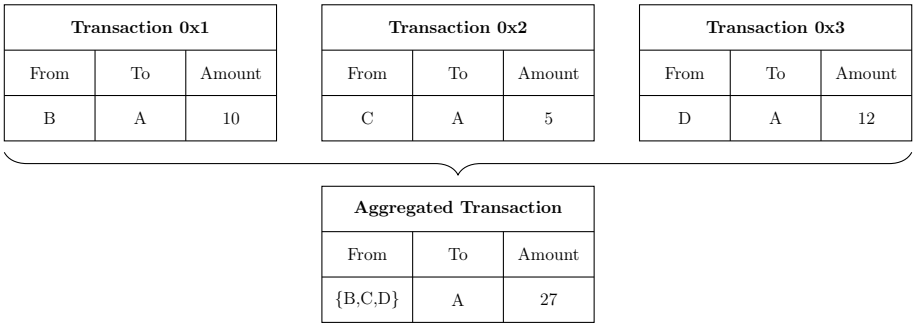
In transaction aggregation by sender, transactions sent *from* a particular user are summed up, shown in Figure 3.6a. On the contrary, in transaction aggregation by receiver, transactions sent *to* a particular user are summed up, shown in Figure 3.6b. Furthermore, we distinguish between transaction aggregation performed by an aggregator or by a leader.

Firstly, transaction aggregation is performed by validators with the role *aggregator*. An aggregator is elected per block height and per shard in the same way a leader is elected with the exception that the *Role* property of Equation 3.4 must be set to "A". In rare cases, there can be zero or more aggregators per block height. If more than one validator fulfills the PoS condition 3.4, the validator who has the lowest value below *Target* becomes an aggregator. Also note that an elected aggregator is not assigned to a random shard. An aggregator only aggregates transactions of the shard they are assigned to, because block bodies containing transactions with self-contained proofs are only stored by validators of the shard they are assigned.

After becoming an aggregator, transactions from previous blocks are aggregated by sender or receiver. Aggregated transactions are then filled into a special



(A) Transaction aggregation by sender.



(B) Transaction aggregation by receiver.

FIGURE 3.6: Transaction aggregation types

type of block called *aggregation block*. An aggregation block of shard s at block height h serves as a proposal to a validator who becomes a leader of shard s at block height $h + 1$. Informally, the following algorithm is performed by each aggregator at block height h:

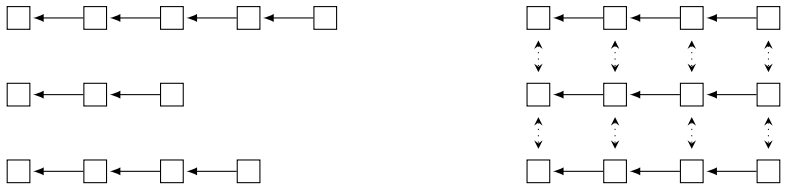
1. Create an empty aggregation block.
2. Pick transactions in two ways:
   (a) For each shard block or aggregation block b, where $Height(b) \leq h - 1$, pick transactions where each sender or receiver appears at least twice and where $Aggregated = false$. Note that $FundsTx$ as well as $AggTx$ can be aggregated.
   (b) For each shard or aggregation block b, where $h - Height(b) \geq length_{ag}$, pick all transactions which have not been aggregated yet, i.e., where $Aggregated = false$.
3. Sum up the amount of coins that have been sent or received, generate the transaction hash, create the aggregated transaction and add it to the aggregation block. Set $Aggregated \leftarrow true$ to transactions in the block they were picked from.

Secondly, transaction aggregation is performed by validators with the role *leader*. When a leader creating a block picks transaction from the local transaction pool, transactions of the same sender are aggregated to a single aggregated transaction. This task is crucial in order to prevent fraudulent proofs, evaluated

in Section 5.1. A leader is also responsible for the verification at height h of an aggregation block that has been proposed at block height $h-1$. The verification process is performed in the same fashion as Bloom filter checks of self-contained proofs.

3.3.4 Block Height Synchronisation

The leader assignment process described in Section 3.2.2 does not guarantee to have one leader per shard. As a result, a shard could have no leader at one block height resulting in a divergence of block heights among different shardchains. For this reason, the block height of shardchains must be synchronized with each other to ensure equally growing chains of blocks.



(A) Without Synchronisation (B) With Synchronisation

FIGURE 3.7: Shardchains could diverge during due to network latency or other reasons resulting in different block heights.

Validators receive blocks and have to perform at least one comparison operation per block in order to decide whether the block belongs to their shard or not. Hence, they can keep track of the number of blocks received at block height h. If the number equals to *NofShards*, validators can be certain that each shardchain of the network grows equally at the same pace. Informally, the following algorithm is performed by each validator v upon receiving block b:

1. The validator checks if the block has already been processed by using the hash of the block and comparing it with the local database. If the block has not been processed yet, continue. Otherwise, the block has already been processed or deemed invalid and the algorithm stops.

2. If $Height(b) = h$, where $Height$ is a function that returns the height of block, continue. Otherwise, the block does not belong to the current block height the algorithm stops.

3. Where *ShardOfBlock* is a function that returns the shard identifier for which a block was created for, and where *ShardOfValidator* is a function that returns the shard identifier the validator belongs to,

   (a) if $ShardOfBlock(b) = ShardOfValidator(v)$, locally store the block header and block body. For each transaction hash included in the block, the validator queries the transaction from the local transaction pool and deletes it, otherwise,

   (b) if $ShardOfBlock(b) \neq ShardOfValidator(v)$, locally store only the block header.

   In either case, increment *ReceivedBlocks* by one, where *ReceivedBlocks* keeps track of the number of received blocks for block height h.

4. Check the self-contained proof. If all Merkle proofs of the SCP are valid, redistribute the block immediately. Otherwise, stop the algorithm.

5. If $ReceivedBlocks = NofShards$, set $ReceivedBlocks \leftarrow 0$ and start the leader election process, as explained in Section 3.2.1. Otherwise, repeat algorithm on arrival of the next block.

3.4 Epoch Finality

An epoch finishes after a predefined number of blocks, called *epoch length*, denoted $length_{ep}$. As pointed out in Section 3.3.4, shardchains can grow in a non-deterministic way if two or more leaders are assigned to the same shard, but since shardchains are synchronized among each other, the finalization of an epoch is guaranteed to end almost at the same time.

Once $length_{ep}$ is reached, validators agree on an epoch block. This special type of block only serves as checkpoint. The network is then repartitioned based on the number of users $NofUsers$. An epoch block is not distributed because it is non-interactively created by every validator in the network. Each validator is able to derive the same hash based on previous blocks.

Chapter 4

Implementation

This chapter explains how the existing Bazo blockchain is revised and extended. Parts of the descriptions are taken from previous papers [3, 5, 24] and updated where necessary.

4.1 Transactions

4.1.1 Stake Transaction (StakeTx)

A node wishing to participate in staking can publish a stake transaction. The transaction contains the commitment key pk_{comm}. The StakeTx consists of the following properties:

Fee A fee that has to be paid for the validator.

Is Staking A boolean value that indicates whether the node wants to join or leave the set of validators.

Account The hash of the public key of the issuer.

Key Commitment When a node wants to join the set of validators it must create a pair of public and private keys (pk_{comm}, sk_{comm}) using RSA. Note that this key-pair is different to the key-pair of the user's wallet. By submitting the public key pk_{comm}, the node commits to this key and it cannot change it in a later step. The (pk_{comm}, sk_{comm}) key-pair will be used in the PoS condition 3.4.

Signature The signature serves the purpose of authentication. The node digitally signs the transaction with its private key sk_{wall}.

4.1.2 System Parameters (ConfigTx)

System parameters can be adjusted with this type of transaction without the need of a hard fork. The transaction must be signed by a root account in order to be accepted. Parameters to change are for example minimum staking amount, minimum waiting time, accepted time difference, slashing window size, or slashing reward.

4.1.3 Funds Transaction (FundsTx)

Users can transfer funds from one account to another by using a FundsTx. In fact, transferring funds is the process of subtracting an amount of coins *from* the sender's account and adding the same amount of coins *to* the receiver's account.

Amount The amount of Bazo coins to be transferred between accounts.

Fee A fee that has to be paid for the validator.

Transaction Counter The account nonce of the sender's account, prevents replay attacks.

Aggregated Determines whether this transaction has been aggregated or not $(true/false)$.

From The public address pk_{wall} of the sender.

To The public address pk_{wall} of the receiver.

Self-contained Proof An ordered list of Merkle proofs referring back to the last epoch block to concisely prove that the sender of the transaction has enough funds to execute the transaction.

Signature The signature serves the purpose of authentication. The node digitally signs the transaction with its private key sk_{wall}.

4.1.4 Aggregation Transaction (AggTx)

An aggregation transaction coalesces two or more $FundsTx/AggTx$. The amount that is transferred in an aggregation transaction is the sum of all $FundsTx/AggTx$. The transaction hash of an aggregation transaction is generated with a Merkle tree whose root node represents the hash of the combined hash of its children node, and the hash of its children nodes are the hash of the combined hashes of their children nodes, and so on until the leaves, where each leaf either represents the transaction hash of a $FundsTx$ or $AggTx$. Figure 4.1 shows the Merkle tree for generating the aggregation transaction hash, where $F = FundsTx$ hash, $H[1/2]$ = intermediate hash of two Fs, and where $A = AggTx$ hash.

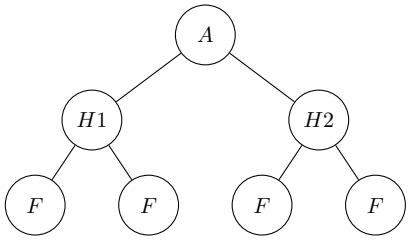


FIGURE 4.1: Generating the hash of an aggregated transaction

Hash The transaction hash acts as a unique identifier of this aggregated transaction. The generation of the hash is shown in figure 4.1.

Aggregated Determines whether this transaction has been aggregated or not $(true/false)$.

Amount The summed up value of all aggregated transaction amounts.

Transaction Counter The value of the highest transaction counter of the aggregated transactions.

From The public addresses [pk_{wall}] of the senders.

To The public addresses [pk_{wall}] of the receivers.

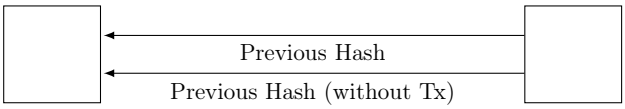
Number of Transactions The number of transactions that have been aggregated.

Hash Data FundsTxs The hashes of all Funds Transactions (FundsTxs) included in this block in sequential order.

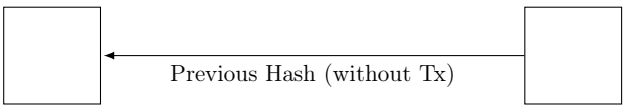
4.2 Blocks

Sharding introduces two new blocks to the system, namely epoch block and aggregation block. Multiple shardchains are running in parallel and a block of a shardchain is referred to as shard block.

Furthermore, with the introduction of transaction aggregation, validators must be able to empty blocks where all transactions have been aggregated. In that case, the pointer to the hash of a previous block would is deemed invalid. In order to keep the integrity of the blockchain, a second hash that points to the previous block without transactions is added to every block.



(A) Before emptying a block



(B) After emptying a block

FIGURE 4.2: Transaction aggregation results in a double-linked blockchain.

Figure 4.2a shows two blocks where one blocks points to the previous block with two hashes. A block which only contains aggregated transactions can be emptied, resulting in an invalid previous hash. However, the second hash that points to the previous block without transactions is still valid.

4.2.1 Shard Block

A shard block is almost identical to a block introduced in Bazo before. The header of a shard block consists of the following properties:

Hash The block hash acts as a unique identifier of blocks within the blockchain.

Previous Hash This value is equal to the identifier of the previous block in the blockchain.

Previous Hash (w/o Tx) This value is equal to the identifier of the previous block without transactions in the blockchain. If the previous block is an epoch block, this property equals to **Previous Hash**.

Number of Bloom filter Elements The number of elements that are in the bloom filter.

Bloomfilter The bloom filter can be queried with pk_{wall} whether the block contains a transaction of pk_{wall} or not. With a false-positive rate of about 10%, the size of the bloom filter is linearly increased or decreased to meet this target.

Number of Users In Shard The number of users in this shard.

The body of a shard block consists of the following properties:

Time in Seconds (Nonce) The number of seconds that a validator needed in order to fulfill the PoS condition.

Timestamp Refers to the block creation time (seconds elapsed since January 1, 1970 UTC).

Merkle Root The value of the merkle tree's root node. Note that the transactions, i.e., the leaves of the Merkle tree, are ordered ascending before generating the Merkle root.

Beneficiary The address hash of the account that receives fee payments and the block reward.

Commitment Proof This property stores a signed message of the $Height$ that this block was created. In particular, $RSA(sk_{comm}, \text{SHA3-512}(BlockHeight))$ where sk_{comm} represents the private key that corresponds to the public commitment key pk_{comm} that was set in the initial StakeTx of the node. Other validators can use pk_{comm} to verify the proof.

Staking Proof The proof that the creator, i.e., the leader of this particular block is eligible to create it. This proof is similar to the self-contained proof of a FundsTx, as mentioned in Section 4.1.3.

Height The height of a block refers to the number of previously appended blocks to the blockchain.

Shard ID The identifier of the shard this block was created for.

Slashed Address A validator can submit a slashing proof when appending a block, i.e., it holds the address of the misbehaving node that must be punished.

Two Conflicting Block Hashes These two properties exhibit the block hashes where the same node has appended a block on two competing chains within the slashing window size.

Number of FundsTxs/AccTxs/ConfigTxs Corresponds to the number of transactions of each type that are included in the block.

Hash Data FundsTxs/AccTxs/ConfigTxs The hashes of all transactions included in this block in sequential order.

4.2.2 Epoch Block

Opposed to shard blocks, which are created by a leader and propagated through the network to every node, epoch blocks are created non-interactively, i.e., every validator locally creates an epoch block without redistributing it based on local information. The header of an epoch block consists of the following properties:

Hash The block hash acts as a unique identifier of blocks within the blockchain.

Previous Hashes This value is equal to the identifier of the previous blocks of every shard.

Previous Hashes (w/o Tx) This value is equal to the identifier of the previous blocks of every shard without transactions.

The body of an epoch block consists of the following properties:

Timestamp Refers to the block creation time (seconds elapsed since January 1, 1970 UTC).

Merkle Root The value of the Merkle tree's root node.

Height The height of a block refers to the number of previously appended blocks to the blockchain plus one.

4.2.3 Aggregation Block

An aggregation block contains aggregated transactions of a particular shard and is created by a validator with the role *Aggregator*. An aggregation block for shard s can only created by a validator of shard s. Furthermore, an aggregation block at block height h serves as a proposal to a leader at block height $h + 1$. The header of an aggregation block consists of the following properties:

Hash The block hash acts as a unique identifier of blocks within the blockchain.

Previous Hash This value is equal to the identifier of the previous block in the blockchain.

Previous Hash (w/o Tx) This value is equal to the identifier of the previous block without Transactions in the blockchain.

Number of Bloom filter Elements The number of elements that are in the bloom filter.

Bloom filter The bloom filter can be queried with pk_{wall} whether the block contains a transaction of pk_{wall} or not. With a false-positive rate of about 10%, the size of the bloom filter is linearly increased or decreased to meet this target.

The body of a shard block consists of the following properties:

Time in Seconds (Nonce) The number of seconds that a validator needed in order to fulfill the PoS condition.

Timestamp Refers to the block creation time (seconds elapsed since January 1, 1970 UTC).

Merkle Root The value of the merkle tree's root node. Note that the transactions, i.e., the leaves of the Merkle tree, are ordered ascending before generating the Merkle root.

Beneficiary The address hash of the account that receives fee payments and the block reward.

Commitment Proof This property stores a signed message of the $Height$ that this block was created. In particular, $RSA(sk_{comm}, \text{SHA3-512}(BlockHeight))$ where sk_{comm} represents the private key that corresponds to the public commitment key pk_{comm} that was set in the initial StakeTx of the node. Other validators can use pk_{comm} to verify the proof.

Staking Proof The proof that the creator, i.e., the leader of this particular block is eligible to create it. This proof is similar to the self-contained proof of a FundsTx, as mentioned in Section 4.1.3.

Height The height of a block refers to the number of previously appended blocks to the blockchain.

Shard ID The identifier of the shard this block was created for.

Number of AggTxs Corresponds to the number of aggregation transactions that are included in the block.

Hash Data AggTxs The hashes of all transactions included in this block in sequential order.

Chapter 5

Evaluation

In the introduction of this paper, we stated that blockchains are subject to an impossible trinity between decentralization, scalability and security. As a result, only two out of these three properties can be achieved to a highest degree and our design of a scalable blockchain may be no exception. Sharding in permissionless, decentralized network with the presence of Byzantine adversary not only results in very new attack scenarios, but also reinstates problems that seem to be solved in non-sharded blockchains.

5.1 Security Considerations

In general, blockchain always raises a variety of pressing security considerations which directly influence the design of it. Each change request could potentially break the entire system and puts the blockchain and its users at risk. Reducing the opportunities for attackers to exploit a potential weak spot or vulnerability requires to minimise the attack surface and applying a structured approach to threat scenarios during design.

With the introduction of self-contained proofs, a validator who becomes a leader for another shard does not need to have knowledge of its users or shard-chain history. Self-contained proofs undoubtedly prove to a validator that a user is eligible to perform a transaction, i.e., user A has sufficient funds to send amount x to B. Hence, a validator must be absolutely capable of deciding whether a transaction is valid or not, having only the last state of the epoch and the transaction itself. A self-contained proof must include a Merkle proof for each transaction transmitted since the last epoch, that is,

$$length([Merkle\ proofs]) = (nonce_{current} - nonce_{epoch}), \tag{5.1}$$

where $[Merkle\ proofs]$ is a list of Merkle proofs, where $nonce_{current}$ denotes the current transaction counter of the account, and where $nonce_{epoch}$ denotes the transaction counter of the account at the beginning of the epoch. Self-contained proofs are unquestionably a hotspot for exploitation, and every possible attack scenario must be taken into consideration.

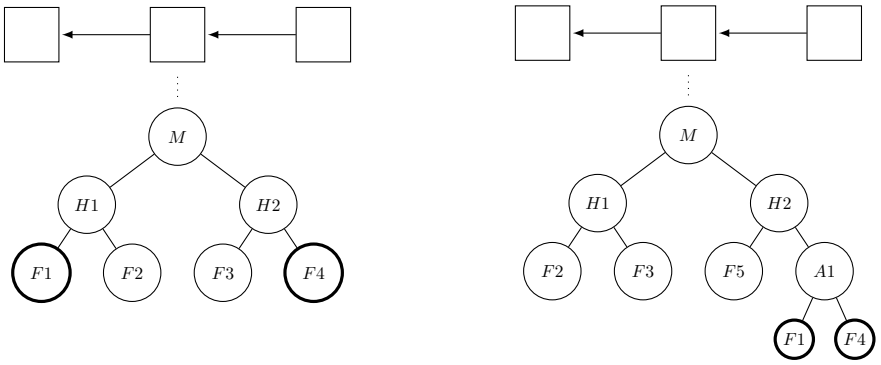
To verify the validity of a SCP, a leader starts by performing block-wise Bloom filter checks. In short, the leader starts at block height $h-1$ and queries the Bloom filter with the user's wallet address pk_{wall}. If the Bloom filter returns *false*, the leader continues with block at height $h-2$ and so on until the Bloom filter returns *true*. If the Bloom filter returns *true*, the leader generates the Merkle root M_1 with the first Merkle proof contained in the SCP and compares

it with the Merkle root M_2 contained in the block. If M_1 does not equal M_2, the SCP is invalid because the first Merkle proof does not correspond to the most recent transaction of the user. If M_1 equals M_2, the leader continues repeats the process for the next Merkle proof until all Merkle proofs have been validated.

The validation process of a SCP works well when there is only one transaction per user and per block. However, an adversarial user can create a fraudulent proof if there are more transactions per user in a block. Essentially, each block contains a Bloom filter which returns *true* when the block contains a particular transaction. If a block contains more than one transaction of the same user, the Bloom filter returns *true* but does not return the number of transactions that are included in the block, which then can be exploited with a fraudulet, yet valid Merkle proof. Example 5.1 demonstrates how this potential vulnerability can be exploited and how it is solved.

Example 5.1. Consider a Merkle tree as shown in Figure 5.1a. Assume that transactions $F1$ and $F4$ belong to the same user. Querying the Bloom filter for this block returns *true*, however, it does not return the number of transactions that are contained within the Merkle tree. The adversarial user can create a valid Merkle proof by providing the values $\{F1, F2, H2\}$ without mentioning $F4$.

Transactions aggregation introduced in Section 3.3.3 mitigates this weakness. A leader aggregates transactions by sender when including them in a block. Querying the Bloom filter for this block returns *true*, because the block contains an aggregated transaction. Figure 5.1b shows the same block as before, but this time, $F1$ and $F4$ are aggregated into $A1$. A user has to provide the values $\{H1, F5, A1\} = \{H1, F5, F1, F4\}$ to generate a valid Merkle proof, since it is computationally infeasible to guess the transaction hash of $A1$ without knowing the hashes of $F1$ and $F4$.



(A) Without transaction aggregation (B) With transaction aggregation

FIGURE 5.1: Extending the Merkle tree with aggregated transactions.

5.2 Attack Scenarios

In non-sharded blockchains, double spending is solved with the nonce. A nonce is just a sequential number tied to every transaction that represents the number of transactions the sender account has made on the network. If a transaction

with a nonce that has already been transacted is submitted, it will be rejected by the network.

In the design of our sharded blockchain, validators neither store blocks of shards other than their assigned shard nor share user information such as funds and transaction counter with each other. For example, if user A of shard 1 sends amount x to user B of shard 2, validators of shard 2 have no knowledge if user A's balance is greater or equal than x or not.

Assuming that the block height between shards is not synchronised, an adversary could take advantage and leave out Merkle proofs of transactions.

Example 5.2. Consider a system as shown in Figure 5.2 with $NofShards = 3$, block height of shard 1 is $h_1 = 5$, block height of shard 2 is $h_2 = 3$, and block height of shard 3 is $h_3 = 4$. Furthermore, assume that an adversarial user A of shard 1 with balance 100 has transferred 5 in transaction $Tx1$ and 95 in transaction $Tx2$ resulting in $balance = 0$ for user A. Next, A could attempt to send 95 to a user B in shard 2 by creating a transaction without including a Merkle proof for $Tx2$. Since validators of shard 2 have no knowledge about blocks in shard 1, they have no way of proving the validity of the transaction from A to B.

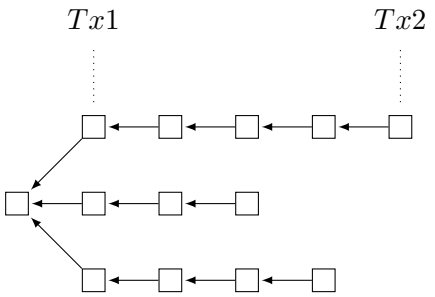


FIGURE 5.2: Diverging shardchains are prone to double spending attacks by adversarial users.

Example 5.2 shows the necessity of block height synchronisation introduced in Section 3.3.4. In order for validators to detect malicious attempts by users, a validator relies on the block height being synchronised amongst shardchains. Then, a validator can perform Bloom filter checks to find out if proofs for transactions are missing in a self-contained proof.

Chapter 6

Summary, Conclusion, and Future Work

In this paper we have presented a sharding concept for the Bazo blockchain that allows the network to process transactions in parallel and significantly increase the transaction throughput. While the purpose of this thesis was only of conceptual matter, our research shows a viable approach towards blockchain scalability and will be of fundamental importance for the next step: the *actual* implementation.

Given the non-exhaustive list of security considerations and attack scenarios in Chapter 5, future work includes more research in exploitable weak points, careful implementation and strict reviews before running a *sharded* version of Bazo in production. For example, a validator could fulfill the leader election equation 3.4 for an invalid block height with a lower $Target$ value than the proposed shard block, resulting in a fork. Then, the validator gradually reduces the difficulty while proposing valid blocks using PoW in order to overtake the non-adversarial shardchain.

Future work further includes studies about the sizes of blocks and shards doing theoretical calculations or running experiments to optimally scale out the blockchain. Preemptive installments for data loss in the event of a complete shard going offline must be thought out. More research must be done about the number of stored blocks after all transactions have been aggregated and what implications smart contracts have in a future implementation.

List of Figures

Bibliography

[1] Anthony Akentiev. *Plasma in 10 minutes*. URL: https://medium.com/chain-cloud-company-blog/plasma-in-10-minutes-c856da94e339 (visited on 07/26/2018).

[2] Gavin Andresen. *Making Decentralized Economic Policy*. URL: https://github.com/bitcoin/bips/blob/master/bip-0101.mediawiki (visited on 03/12/2018).

[3] Simon Bachmann. *Proof of Stake for Bazo*. Ed. by Thomas Bocek. 2018.

[4] Bitcoin. *Block size limit controversy*. URL: https://en.bitcoin.it/wiki/Block_size_limit_controversy (visited on 03/07/2018).

[5] Roman Blum. *Cryptographic Sortition for Proof of Stake in Bazo*. Ed. by Thomas Bocek. 2018.

[6] brainbot. *Raiden Network - Fast, cheap, scalable token transfers for Ethereum*. URL: https://raiden.network (visited on 04/24/2018).

[7] Cryptocurrency Market Capitalizations. *Bitcoin Market Capitalization*. URL: https://coinmarketcap.com/currencies/bitcoin/historical-data/?start=20180816&end=20180816 (visited on 08/17/2018).

[8] Elasticsearch. *Sharding*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html#getting-started-shards-and-replicas (visited on 03/12/2018).

[9] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, et al. "Bitcoin-NG: A Scalable Blockchain Protocol". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 45–59. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal.

[10] Ethereum Foundation. *Blockchain Trilemma*. URL: https://github.com/ethereum/wiki/wiki/Sharding-FAQ#this-sounds-like-theres-some-kind-of-scalability-trilemma-at-play-what-is-this-trilemma-and-can-we-break-through-it (visited on 03/07/2018).

[11] Jeff Garzik. *Making Decentralized Economic Policy*. URL: http://gtf.org/garzik/bitcoin/BIP100-blocksizechangeproposal.pdf (visited on 03/12/2018).

[12] Garrett Hardin. "The Tragedy of the Commons". In: *Science* 162.3859 (Dec. 1968), pp. 1243–1248. URL: http://www.sciencemag.org/cgi/content/full/162/3859/1243.

[13] Visa Inc. *Visa Inc. Facts & Figures, Nov 2015*. URL: https://usa.visa.com/dam/VCOM/download/corporate/media/VisaInc_factsheet_11012015%20(002).pdf (visited on 03/07/2018).

[14] Michael Epstein Andrew Fikes Christopher Frost J. J. Furman Sanjay Ghemawat Andrey Gubarev Christopher Heiser Peter Hochschild Wilson Hsieh Sebastian Kanthak Eugene Kogan Hongyi Li Alexander Lloyd Sergey Melnik David Mwaura David Nagle Sean Quinlan Rajesh Rao Lindsay Rolig Yasushi Saito Michal Szymaniak Christopher Taylor Ruth Wang James C. Corbett Jeffrey Dean and Dale Woodford. "Spanner: Google's Globally-Distributed Database". In: (Aug. 2013).

[15] Aggelos Kiayias, Alexander Russell, Bernardo David, et al. *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol*. Cryptology ePrint Archive, Report 2016/889. `https://eprint.iacr.org/2016/889`. 2016.

[16] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, et al. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *2018 IEEE Symposium on Security and Privacy (SP)*. Vol. 00, pp. 19–34. URL: `doi.ieeecomputersociety.org/10.1109/SP.2018.000-5`.

[17] Selena Larson. *Bitcoin split in two, here's what that means*. URL: `http://money.cnn.com/2017/08/01/technology/business/bitcoin-cash-new-currency/index.html` (visited on 03/07/2018).

[18] Loi Luu, Viswesh Narayanan, Chaodong Zheng, et al. "A Secure Sharding Protocol For Open Blockchains". In: CCS '16 (2016), pp. 17–30. URL: `http://doi.acm.org/10.1145/2976749.2978389`.

[19] Microsoft. *Sharding pattern*. URL: `https://docs.microsoft.com/en-us/azure/architecture/patterns/sharding` (visited on 03/12/2018).

[20] MongoDB. *Sharding*. URL: `https://docs.mongodb.com/manual/sharding/` (visited on 03/12/2018).

[21] Gordon E. Moore. "Readings in Computer Architecture". In: ed. by Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming More Components Onto Integrated Circuits, pp. 56–59. URL: `http://dl.acm.org/citation.cfm?id=333067.333074`.

[22] Joseph Poon and Vitalik Buterin. *Plasma: Scalable Autonomous Smart Contracts*. URL: `https://lightning.network/lightning-network-paper.pdf` (visited on 04/24/2018).

[23] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. URL: `http://plasma.io/plasma.pdf` (visited on 04/24/2018).

[24] Livio Sgier. *Bazo - A Cryptocurrency from Scratch*. Ed. by Thomas Bocek, Bedrija Hamza, and Bruno Rodrigues. 2017.

[25] Jason Spasovski and Peter Eklund. "Proof of Stake Blockchain: Performance and Scalability for Groupware Communications". In: *Proceedings of the 9th International Conference on Management of Digital EcoSystems*. MEDES '17. Bangkok, Thailand: ACM, 2017, pp. 251–258. URL: `http://doi.acm.org/10.1145/3167020.3167058`.

[26] Zhe Zilliqa Team. *The Zilliqa Technical Whitepaper*. URL: `https://docs.zilliqa.com/whitepaper.pdf` (visited on 07/10/2018).