University of Zurich^UZH

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Evaluation and Improving Scalability of the BAZO Blockchain

Fabio Maddaloni
Zurich, Switzerland
Student ID: 15-703-150

Supervisor: Sina Rafati
Date of Submission: April 23, 2019

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

English

Blockchains are one of the newer and recently often discussed topics in the information and communication technology world. Therefore, many research projects are currently ongoing on this subject. Every new idea reaches a point where performance and usability are compared to existing technologies. Additionally, different versions of the same research project are compared against each other with the goal to estimate, if the applied improvements bring the desired outputs.

Cryptocurrencies are based on the blockchain technology because under certain conditions they are extraordinary secure. This is attributed to the decentralization and validation mechanisms of a blockchain and the user's anonymity. Since blockchains were, especially at the beginning, often used for currencies, it is not surprising that many blockchain developers and their projects choose the biggest credit card companies as their comparison target in terms of transaction speed.
At the time of writing this thesis hardly any blockchains is able to even process nearly as many transactions per second as credit card companies. Therefore, a part of this thesis is devoted to a scalability improvement.

The scalability improvement is an aggregation process of transactions with either the same sender or receiver. Thus, depending on the use case, fewer transactions need to be written into a block and therefore, the number of transactions per second can be increased. The implementation of the scalability improvement is realized onto the BAZO blockchain's source code.

Furthermore, the performance before and after the improvement is tested in a newly created global network based on 20 miners. The results are evaluated and can be used to plan the next steps to get even higher transaction speeds.

Deutsch

Blockchains sind eines der grossen neuen Theme in der Informations- & Kommunikationstechnik und genau deshalb gibt es viele Forschungsprojekte dazu. Jede neue Idee kommt an einen Punkt, bei dem die Leistung und Verwendungsvor- sowie -nachteile mit bereits bestehenden Technologien verglichen werden. Oft werden verschieden Versionen einer neuen Technlologie verglichen, um abzuwägen, ob die angebrachten Verbesserungen auch die gewünschten Effekte zur Folge haben.

Kryptowährungen basieren auf der Blockchain Technologie, da diese, unter bestimmten Voraussetzungen, besonders sicher ist. Dies ist vorallem auf die Dezentralisierung und die Validierung der Blockchain zurückzuführen, sowie der Anonymität der Benutzer zu verdanken. Da viele Blockchain Projekte, vorallem als die Technologie ganz neu war, für Währungen verwendet wurden, ist es nicht sonderlich erstaunlich, dass viele Entwickler und ihre Projekte den grossen Kreditkartenfirmen, in Hinsicht auf die Anzahl Transakionen pro Sekunde, nacheifern.
Als diese Arbeit verfasst wurde, waren die meisten Blockchains nicht in der Lage annähernd hohe Transaktionsraten wie die Kreditkartenbetreiber zu erzielen. Aus diesem Grund ist ein Teil dieser Arbeit einer Lösung zum Skalieren von Blockchains gewittmet.

Die gewünschte Skalierbarkeit wird durch Zusammenfassen von bestimmten Transaktionen mit dem gleichen Sender oder Empfänger gemacht. Je nach Anwendungsfall, können deutlich mehr Transakionen in einen Block geschrieben werden und dadurch die Anzahl Transakion pro Sekunde erhöht werden. Die Verbesserung wird anhand einer Implementation basierend auf der BAZO Blockchain durchgeführt.

Weiter wird die Leistung vor und nach der Verbesserung in einem globalen, neu erstellten Netzwerk mit 20 Miner getestet, analsiert und verbessert. Die Resultate werden begutachtet sowie ausgewertet und können daher als Grundlage zur Planung der nächsten Schritte zum Erreichen von noch höheren Transaktionsgeschwindigkeiten gebraucht werden.

Acknowledgments

I would like to thank my supervisor, Sina Rafati, for his continuous assistance and inputs during the last six months. It was a tremendous relief that I could write to him or show up in his office any time and received always a precious answer. This ensured a continuous work, which is highly appreciated.

Also, I would like to thank Prof. Dr. Thomas Bocek for his great inputs and wider perspective on the project during the weekly meetings. Most often, these meetings were truly eye-opening and I, as a student and personally, learned a lot.

Furthermore, I thank Kürsat Aydinli, Roman Blum and Marc-Alain Chételat for their valuable answers regarding code-questions.

And last, I would also like to thank Prof. Dr. Burkhard Stiller, the head of the Communication Systems Research Group, for the possibility to work on such an amazing and interesting project.

Contents

Chapter 1

Introduction

1.1 Motivation

As with every new technology, performance metrics are important evidences if a novel approach is usable at large scale or if the idea will stay a niche product. Often new approaches are compared against older, already established techniques, technologies and products. This should bring a new perspective on what is truly possible and how it should be improved to reach the desired goals. Nevertheless, it is important that similar things are getting compared. Often also different versions of a product are compared against each other to see if new applied solutions bring the desired effects.

One of the biggest problems of blockchains is scalability. Because of the decentralization and its unique consensus approach, it is hard to scale a blockchain. One goal of the whole blockchain world is to achieve transaction rates comparable to the biggest credit card providers. Currently, this is not possible and therefore multiple research projects are running on different blockchains [24].

1.2 BAZO - The Blockchain

BAZO is a research blockchain created and mainly developed at the University of Zurich in 2017. At the beginning, it was a private/invite-only blockchain which was planned to replace a traditional bonus system of a credit card issuer [44]. BAZO used a Proof-of-Work consensus algorithm but soon it changed to a Proof-of-Stake version. This change was performed in early 2018 [19].

Since then, BAZO became more evolved and, at the time of writing, various research projects and improvement attempts in several directions from different universities are in progress. Some of them are going into a direction of changing BAZO from a bonus system replacement to a blockchain specialized for the Internet of Things (IoT). One overlapping goal of all those projects is to scale the blockchain to be able to handle more transactions per second.

1.3 Description Of Work

This thesis consists of three parts which overlap slightly and therefore also influence each other.

The objective of part one is the creation of a global network of BAZO miners and clients. This network should run on virtual machines hosted by various cloud providers. It is used for performance tests with the ability of geographical distance between peers as well as tests with multiple miners.

Furthermore, a scalability improvement, namely *transaction aggregation*, for the BAZO blockchain is implemented and evaluated. In this paper the core mechanisms, design ideas and difficulties are discussed. In the evaluation part, the scalability mechanism is analyzed and interpreted as well as compared to older BAZO versions.

Besides these two planned core parts, a third part, where all bugs and its fixes which were found during the process of the thesis, are described and developed solutions explained.

1.4 Thesis Outline

The thesis starts with a small introduction in chapter 1 followed by a chapter comparing different scalability ideas and approaches. This is handled in chapter 2. Chapters 3, 4 and 6 do cover the main aspects of this thesis, namely the network preparation and setup as well as the scalability improvement. All bugs found during the process are listed in chapter 5. There the problems and the elaborated solutions are described and elucidated. This thesis ends with a small summary and conclusion in chapter 7.

Chapter 2

Related Work

This chapter describes related work for the two core directions of this thesis, *performance analysis* and *scalability improvements.*

2.1 Performance Analysis

Blockchains are often compared against each other in term of performance. To be more precise, most of the time they are compared in regards to transaction throughput. Many people subsequently assume that a faster blockchain is better than a slower one. This assumption can sometimes be problematic because many blockchain projects reach high throughput at the cost of reducing and neglecting important characteristics.

Currently VisaNet, one of the biggest electronic payment systems, can handle over 24'000 transactions per second (*TPS*) as listed on their US-website [12]. However, on the Swiss website, they claim up to 56'000 *TPS* are possible with *VisaNet* and in the official fact sheet linked on the same website, *VisaNet* should be able to process 65'000+ transactions per second [8,13]. No matter if these numbers are correct or not, they are only peak values and not handled over a long time period. Especially blockchain near people often say that *Visa* is able to process a few thousand transactions per second, but never numbers like those officially published ones [34, 43]. This is a great example, that *TPS*, despite the fact that it is probably the most used and widespread metric as soon as transactions are involved, should always be used with care.

Although it is not 100% sure how many transactions *VisaNet* can handle per second, it is a clear goal for many blockchains to be faster than *Visa*. Also here multiple different rankings exist. This is probably caused by the fact that new blockchain projects sprout in a fast manner. According to a Deloitte study, published in September 2018, alone on Github around 6'500 active blockchain projects are available [29].

Howmuch.net created a ranking, based on transactions per second, for blockchains and its competitors in 2018 and according to this, *Ripple* is second fastest after *Visa* in regards to transactions per second. *Ripple* does reach 1'500 *TPS* and therefore, nearly 7.8

times more Transactions than *PayPal* which is listed at 193 *TPS*. The two most famous cryptocurrencies, *Bitcoin* and *Ethereum*, can handle around 7 resp. 20 *TPS* [41]. *Zilliqa*, another blockchain project but not included in this ranking, claims that they are able to process over 2'800 transactions per second with the help of sharding [15].

In 2018 a team from the University of Sydney created a blockchain called *Red Belly Blockchain*. This blockchain is, at the time of writing, probably the fastest one on earth. The research team claims that their blockchain can handle up to nearly 700'000 *TPS* [9]. This team also used a global network of virtual machines hosted on *Amazon Web Services* (*AWS*).
In their paper they spawn 1'000 virtual machines in 14 data centers and get a final *TPS* of 30'684 [25]. The difference in these *TPS* numbers is probably related to the fact, that at their peak of 700'000 *TPS*, all machines were located in one availability zone whereas in the second experiment were they reached around 30'000 *TPS* they were located globally. However, they do not deliver how they exactly calculate and measured the transactions per second.
They used *c4.large* machines for their so-called requester and *c4.8xlarge* computes for proposers, which behave similarly as miners. These machines have 3.75GB resp. 60GB of RAM and both are optimized for computing-intensive workloads with an extremely high level of data processing. Hence, especially because of the *c4.8xlarge* virtual machines, it is possible to argue about the decentralization of this test run. This problem is further described in section 2.2.1, because it contradicts the concept of a truly distributed blockchain.

2.2 Scalability Improvements

One of the biggest problems the latest blockchains have, is that they cannot be scaled up easily. There is always a trade-off between *security*, *decentralization* and *scalability*. Every blockchain can select two out of these three attributes. The third one will become difficult up to impossible to achieve. This trilemma is the problem of scaling blockchains.

As an example, *Bitcoin* and *Ethereum* were designed with a focus on decentralization and security. The *Red Belly Blockchain* is probably designed with a focus on scalability and speed rather than security and decentralization. In other words, when a super fast blockchain is developed with toady's techniques, it must have some drawbacks in either decentralization or security [30]. That is one reason, why *Bitcoin* and *Ethereum* have much lower *TPS* in comparison to other blockchains.

In the following subsections some blockchain scalability improvements are listed and their individual advantages & disadvantages are shortly stated. Other distributed ledger techniques, like directed acyclic graphs, are not discussed here.

2.2.1 Increase Block Size

Increasing the block size is one simple way to increase the throughput of a blockchain system. This works not only if the bottleneck is the block size. When the block size is somehow limited, only a restricted number of transactions fit in one block and with a given block interval only a limited number of transactions per second can be validated in the network. Thus, one of the easiest ways to enhance the *TPS* is to increase the block size and by doing this rise the maximum of transactions per second [18]. Increasing the block size can, if utilized, only be used as a first step towards *TPS* enhancement because it does bring some problems [27].

With an enhanced block size, the overall blockchain size does grow as well. This can be a problem, for example when a new miner joins the network and has to request all blocks first. In such a situation, but also during daily business as a miner, the hardware on which a miner is running has to be more powerful with regards to the ability of handling larger blocks [18]. When the block size becomes bigger and bigger, at one point only supercomputers can handle them. In that case, the network will probably not be truly decentralized, because a normally powerful computer cannot work as miner anymore. This is exactly the negative connotation of the *Red Belly Blockchain* described in the subchapter 2.1, where they use virtual machines with 60GB of RAM.

Furthermore, it is good to know, that an increment of the block size is done on-chain, also known as layer 1 scalability improvement. This means that the enhancement of the block size is done directly in the code base. Because this is a fundamental change, a hard fork is required (Forks are further explained in section 5.1) [30].

The effects of different block sizes are evaluated and compared in subsection 6.1 in the Evaluation chapter.

2.2.2 Decrease Block Interval

Decreasing the block interval is another simple way to enhance network performance. When mining fixed sized blocks lessening of the timespan between two following blocks does enhance the *TPS* [27].

In this case, alike problems as in subsection 2.2.1 will occur. At one point the data size will become too big, not because block entities are too large, but because of the too small interval between two consecutive blocks. Thus, many blocks are created, which increases the overall amount of data. As a consequence, only powerful computers and miners with a good network connection are fast enough. This probably leads to a decentralized network.

Similar to the increment of the block size, this is an on-chain solution which will bring a hard fork once implemented and rolled out [30].

The effects of different block intervals are evaluated and compared in subsection 6.2 in the Evaluation chapter.

2.2.3 Smaller Transaction Size & Transaction Aggregation

When transaction sizes can be reduced, this can bring the same positive impact as increasing the block size, described in subsection 2.2.1, but without the negative aspect. It is either possible to push more transactions into one block by enhancing the block size or by reducing the transaction size. With the in this thesis described concept two advantages in comparison to the enhanced block size are visible. Firstly, the size of a block must not be increased, meaning the miners do not have to handle large blocks and therefore they must not be extraordinarily powerful. And secondly, less transaction data has to be sent through the network what would obviously reduce the network traffic.

It is definitely a good idea to keep the transaction sizes as small as possible and only include truly necessary data in it. The transaction size is given through the amount of data in a transaction. Therefore, a contradiction between the necessary volume of data inside a transaction and the overall size of a transaction exists. A trade-off between necessary data and the transaction size needs to be found.

BAZO, originally, does already include some kind of scalability improvements because only the transaction hashes are stored inside a block [44]. This enhances the throughput, as described in section 5.2.2.

With transaction aggregation, the transactions get summed up in a special way such that fewer transactions have to be written into one block. It is a concept designed in a way, that miners do not store transactions older than a predefined number of blocks. Furthermore, transactions of the same sender or receiver can be aggregated while mining a new block. A small example would be if A sends 1 coin to B and 3 to C, instead of writing both transactions in a block, only one transaction is directly saved in a block. This transaction has the form: A send 4 coins to B & C. It is done vice versa for receiving coins.
With this technique the overall blockchain size should remain small and at the same time the number of transactions validated per seconds should rise. The concept, elaborated for the BAZO blockchain by Roman Blum, is discussed in further details in section 3.2 [23].

2.2.4 Sharding

Sharding is a concept, where the blockchain deliberately splits into smaller divisions, so-called *shards*, to increase the throughput. It is widely used for databases, where a database is split to several chunks which can be processed in parallel [22,31]. This is kind of a divide & conquer strategy, which is planned to be applied to blockchains with some differences.

In the blockchain world, all open transactions get split into different groups according to, at this point unimportant, rules. Each shard, with a number of miners, then works on his chunk of transactions and therefore, transactions only get validated in one shardchain and not in the whole blockchain [21, 22, 31].

It is a layer 1 scalability improvement, meaning it is done directly on-chain and a hard fork will occur once implemented [30].

There is a sharding concept for BAZO, created by Roman Blum. The intention behind this is to divide the network into different shards. All transactions are stored in every miners mempool. But a miner only validates the transaction if a user, belonging to the same shard, is either sender or receiver in this transaction. After a predefined number of blocks, the shards will congregate to one specific block, called the *epoch block* (denoted with E1 – E4 in figure 2.1). These *epoch blocks* are the same for every miner in the network and thus, all forks between two consecutive *epoch blocks* are resolved. At this *epoch blocks*, the number of shards and the number of miners per shard will be calculated. With a load-balancing algorithm, it is ensured, that the optimal number of shards is found as well as the ideal number of miners per shard [23].

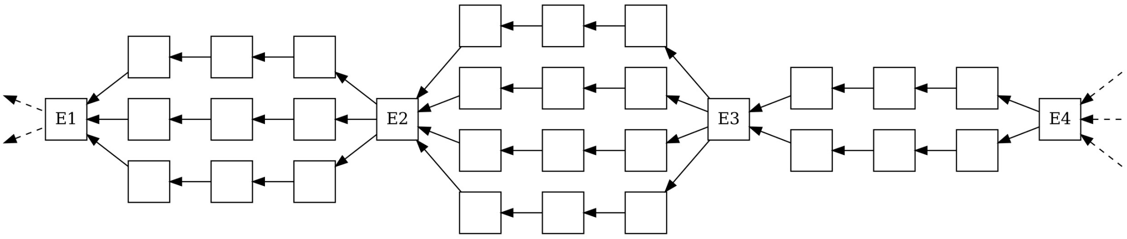


Figure 2.1: Sharding concept with dynamically load-balancing [23]

Sharding pertains as one of the most promising scalability concepts and multiple blockchain projects (*Ethereum* [30], *Elastico* [35] or *RapidChain* [48]) are planning to implement it. On January 31st, 2019, *Zilliqa's* mainnet with sharding went online and demonstrates that sharding works [20].

However, basic sharding does also bring a bigger problem. Since the miners only work in one shard, a 51% attack, with the goal to control one shard, becomes way more easy [17]. As an example, when 12 miners are mining in one network, it needs at least control over seven miners to possess over 51% of the hashing power. When these 12 miners now are split into four shards, an attacker only needs to hold three miners to control one shard. Then, the attacker has influence on which transactions are validated in this shard. In this example, a bad influence to 25% of all miners in the network would be enough to control a shard.

2.2.5 Altcoins ⟷ New Blockchain

Whenever some major changes are introduced to a network, it is up to the user to decide if the new settings and ideas match their own thoughts. When this is not the case, they can search for others with the same conception and try to create a new blockchain with a hard fork [30]. This new blockchain is then an alternative to the other blockchain and therefore it is called an *altcoin* [14].

This is good on one hand because discrepancies about the future of a blockchain get solved easily [32]. But it also brings, at least, two problems on the other hand. Firstly, it is not a true scaling of the blockchain. When the new currency simply is created by users who want a n-times larger block size, similar problems as in subsection 2.2.1 will occur at one point. Thus, this does not solve the scalability problem of the original ledger. Secondly, the blockchain gets securer the more user participate. When a blockchain now forks, the security may be reduced because less user compete.

2.2.6 Off-Chain Solutions

Off-chain solutions are, in contrast to the previously described on chain approaches, built on top of the actual blockchain and only certain transactions are listed in the chain. Often *side-chains* or *state-channels* are used. This should prevent congestion of the network, reduce data traffic and fasten up the blockchain. With this approach, the blockchain base-code does not get changed and a hard fork is not required [30].

An interesting approach is used by the lightning network built on top of *Bitcoin*. It creates *state-channels* between two entities and transactions among these two can be executed through the channel without validating all of them in the blockchain. Simply spoken, the goal of these channels is that only the total, absolute transaction balance is listed in the blockchain. All, but especially the small, transactions among these two entities are aggregated and the total amount then written to the blockchain [39, 40].

An advantage is that all transactions happening inside a channel are only visible to the two entities what results in even bigger anonymity than what cryptocurrencies are already providing. Furthermore, they do not have to wait until a new block is mined and the transactions are validated in there. Also, the entities have lower transaction costs because not all transactions are listed in the blockchain. This is opening more space for other transactions [39].
The biggest disadvantage here is, that these channels are only useful when two entities exchange data on a regular basis because a channel needs to be established whenever two entities want to exchange coins for the first time [39].
At the time of writing the *Lightning Network* can be used with *Bitcoin* and the counterpart for *Ethereum* is called *Raiden Network* [30, 40].

Plasma, another off-chain approach using the *Ethereum* blockchain, uses *side-chains* which originate from the original chain. These *side-chains* will push transactions to the *root-chain* at a certain point. They are also called *child-chains* and can issue other *side-chains* by themselves. Therefore, it can be seen as a hierarchical tree of *side-chains*, which all will transfer data and information to the *root-chain* periodically. With plasma, *Ethereum* should be able to handle larger data sets and therefore a higher throughput should be feasible [42]

2.2.7 Scalable Consensus Mechanisms

There are three main ways of scalable consensus mechanisms: *Delegated Proof-of-Stake*, *Byzantine Fault Tolerance* and *Proof-of-Authority* [30].

The approach of *Delegated Proof-of-Stake* is kind of similar to a democratic process. Here all token holders can vote for delegates which validate transactions. Whenever one delegate does not validate transactions correctly, it can be voted out by the other normal users. Basically, the top n delegates get a regular salary, but they can be voted out any time. This ensures that they do their work in a proper way. But, it also brings centralization, because only these nodes hold and validate the whole blockchain [17, 29]. The voting power of a specific user is dependent on the size of the stake [30,45]. Therefore, it is not a truly democratic process. It is similar to stocks and shareholders. The more shares you have, the more influence you have on the shareholder meeting. However, this does also bring new possibilities of attack scenarios, as an example when smaller miners sell their vote to wealthy or influential miners.

Byzantine Fault Tolerance, a solution to the *Byzantine Generals' Problem*, is listed as a possible scalability improvement. *Hyperledger Fabric* and *Zilliqa* are two projects which use a high-performance version of the Practical Byzantine Fault Tolerance [30]. The developers behind *Zilliqa* claim that they are able to reach slightly more than 2'800 *TPS* with the help of this Byzantine Fault Tolerance and sharding [15, 46].
The *Federated Byzantine Agreement* is another version of the *Byzantine Fault Tolerance* which is used in *Ripple*. In *Ripple* only a chosen list of so-called validators can validate blocks and transactions. This harms the concept of true decentralization [30].

Proof-of-Authority is the third way on how blockchains can scale with respect to the consensus mechanism. This approach does violate the concept of true decentralization and anonymity because there exist entities which are participating as administrators of a network. These administrators are completely identified, sometimes even with the help of authoritative data such as in public notary databases [28, 30]. Therefore, it is more convenient to use this in a private and / or permissioned network.

2.2.8 Comparison And Conclusion

This subsection, especially table 2.1, is shortly summarizing section 2.2 and giving a short overview of the rating of a specific scalability improvement.

As seen in the previous subsections, scaling a blockchain is actually needed to gain a higher throughput and thus gain a wider area of application. But it is not a trivial scaling procedure and more research in this area is needed. There are already some good scaling ideas around, like *Sharding* and *Transaction Aggregation*. Especially when combining these mechanisms higher rates in regards to transactions per second are feasible. However, particularly for these two approaches, the implementation is not as easy as thought, mainly due to the decentralization.

Table 2.1: Scalability ideas, their rating sand reasons therefore.

Scalability Idea	Rating	Reason
Increase Block size	Bad	Not a true scalability improvement and a problem of decentralization because of the amount of data.
Decrease Block interval	Bad	Not a true scalability improvement and a problem of decentralization because of the amount of data.
Transaction Aggregation	Good	This can indeed scale the blockchain because more transactions can be aggregated inside one block.
Sharding	Good	With this scalability improvement, more transactions should be processed due to the parallel like processing.
New Chains	Bad – Medium	At one point there may exists a great number of different chains. These chains may not have many users and are therefore not extremely secure or decentralized.
Off Chain Solutions	Medium – Good	They are good, but often only for some special use cases.
Scalable Consensus Algorithm	Medium	Some of them would help in scaling a blockchain but also bring some side effects like less anonymity, centralization or new attack scenarios.

Chapter 3

Design

In this chapter, design ideas and decisions for the performance analysis and the transaction aggregation are listed and shortly explained.

3.1 Performance Analysis

The goal is to implement a global BAZO network with the help of *Amazon Web Services* (*AWS*) and the *Google Cloud Platform* (*GCP*). Furthermore, a local instance runs at the University of Zurich on server *b04*.

3.1.1 Metrics

The metrics used in the evaluation process are defined as follows [47]:

- **TPS** = *Transactions per second* = Average number of transactions validated per second.
- $\mathbf{TPS_{calc.}}$ = *Transactions per second calculated* = Theoretical maximal number of transactions, which can be validated per second with given block size and block interval.
- $\mathbf{TPS_{sent}}$ = *Transactions per second sent* = Average number of transactions sent to the network per second.
- **ABI** = *Actual block interval* = Timespan between two consecutive blocks.
- **BCS** = *Block chain size* = Size of the blockchain.

Of course, there are other metrics, but at the time of writing the metrics above were decided to be the most interesting ones because they illustrate the possibilities and power of transaction aggregation the most.

TPS

The *transactions per second* are calculated with the help of the timespan between the first transaction sent to the network and the first block which includes all sent transactions, called the *timespanValidation*.

$$\mathbf{TPS} = \frac{numberOfValidatedTransactions}{timespanValidation} = \frac{numberOfValidatedTransactions}{(timeOfBlock - timeFirstTxSent)} \tag{3.1}$$

The *TPS* on its own does only tell how many transactions are validated in a blockchain per second. It does not reveal anything about limiting factors and why a specific *TPS* is reached.

$\mathbf{TPS_{calc.}}$

The maximal possible calculated number of transactions, which can be validated with a set block size and interval, is computed out of the number of transactions which fit into a block and the block interval (in seconds). The blockSize has unit byte.

$$\mathbf{TPS_{calc.}} = \frac{numberOfPossibleTxPerBlock}{blockInterval} = \frac{\frac{(blockSize - 658byte)}{32byte}}{blockInterval} \tag{3.2}$$

This number provides the upper maximum when blocks are validated in the correct user-set interval. Therefore, it is mainly used as a benchmark, since it is basically just a theoretical value. It should always be handled with care because the theoretical and not the actual block interval is taken into account for this. It indicates the maximal speed a blockchain can have without any type of scalability improvements, such as sharding or transaction aggregation.

However, if the *TPS* is very close or similar to the $TPS_{calc.}$ and the TPS_{sent} is far above than the other two, the blockchain is probably limited either through the block size or by the block interval.

$\mathbf{TPS_{sent}}$

The *sent transactions per second* is the average number of transactions sent to the network by all clients and it indicates the upper limit for the number of transactions which can be validated per second.

$$\mathbf{TPS_{sent}} = \frac{numberOfSentTransactions}{timespanSent} = \frac{numberOfSentTransactions}{(timeLastTxSent - timeFirstTxSent)} \tag{3.3}$$

The TPS_{sent} indicates how fast transactions are sent to the network. When the TPS_{sent} and the TPS are close to each other, it can be assumed, that all transactions get validated shortly after they are issued. This because it does not take much longer to validate all transactions than it takes to send them. Thus, if they diverge a lot, it is an indicator, that it takes longer until the transactions are validated.

Furthermore, it is an indicator for the upper TPS limit, since the TPS can not be higher than the TPS_{sent}.

ABI

The *ABI* does reveal the actual timespan between two consecutive blocks. It does indicate if the network satisfies the defined block interval. Adjusting the block interval needs time to become consistent.

$$\mathbf{ABI} = \frac{(endTime - startTime)}{numberOfBlocksInBetween} \tag{3.4}$$

It is simply measured as the difference between two following blocks or as an average with the timespan between two selected blocks (*endTime - startTime*) and the number of blocks between them.

BCS

The *BCS* is the metric for the blockchain's overall size. The block's size consists out of the fixed part and the size of all transactions. Since BAZO only writes the transaction hashes into blocks, the actual transaction size is not taken into account here. Once they get emptied, the overall blockchain size should shrink.

It is calculated as a sum of all block's size when they are secure enough. These blocks are all blocks from the genesis block (*genB*) up to the last validated and secure block (*lvB*). Taking only valid and secure blocks should help to prevent miscalculations of blocks if a rollback scenario appears.

$$\mathbf{BCS} = \sum_{genB}^{lvB} (fixedBlocksize + hashSize * nrOfTransactions) \tag{3.5}$$

It should reveal, if the transaction aggregation, and especially the emptying of blocks, once all transactions are aggregated (more in section 3.2.3), is reducing the blockchain's overall size.

3.2 Transaction Aggregation

This chapter addresses the transaction aggregation, the underlying techniques and its design decisions. The concept and the basic idea is based on the paper of Roman Blum [23].

3.2.1 Idea

The idea behind transaction aggregation is to aggregate valid transactions such that multiple transactions from one sender or to one receiver are visible as one transaction in the blockchain. This should enhance the *TPS* because more transactions can be validated in one block. Thus, the block size becomes less of a limiting factor when many transactions with the same sender or receiver are issued to the network. Furthermore, transaction aggregation reduces the blockchain's overall size because fewer transactions are visible in the blockchain. Especially when aggregating transactions, which are validated in an already closed block, the overall blockchain size can shrink since at a certain point these blocks can be emptied completely. This results in a reduced block size.

An Example: User *A* sends 2 BAZO-Coins to *B* and 5 Bazo-Coins to *C*. Without transaction aggregation both transactions are listed in a block as (schematic) *(A →B : 2)* & *(A →C : 5)*. With transaction aggregation only one transaction *(A →[B,C] : 7)* will be written into the block. This illustrates that the overall blockchain size could be smaller and more transaction can be handled because they are aggregated.

The aggregation is planned to be fully hidden from the users.

3.2.2 Aggregation Of Transactions

For the aggregation, a new type of transactions, called *AggTx*, is introduced. Furthermore, all funds transactions, called *FundsTx*, are slightly updated. The changes are shown below.

FundsTx

Aggregated: The variable 'Aggregated' is a *boolean* and does indicate if this transaction is aggregated already.

Block: In this filed the hash of the block, in which this transaction is validated the first time, gets stored. This is used for rollback scenarios. It is of type *[32]byte*.

AggTx

This type of transaction does aggregate and sum up matching funds transactions and older aggregation transaction. Instead of these transactions, an *AggTx* will be listed inside a block.

Amount: The amount is the summed up amount of all transactions aggregated inside this *AggTx*. It is of type *uint64*.

Fee: The fee of this transaction. It is set to 0 because, at the time of writing, the users should not be charged for this type of transaction. It is also a *uint64*.

From: It is a slice where the addresses of all senders sending a transaction aggregated in this block are stored. It is a slice of type *[][32]byte*

To: This is the counterpart of the *From* field and filled with the addresses of the transactions' receivers. It is also a slice of type *[][32]byte*.

AggregatedTxSlice: This slice is of type *[][32]byte* and does store all hashes of the transactions aggregated inside this *AggTx*.

Aggregated: The variable 'Aggregated' is a *boolean* and does indicate if this transaction is aggregated.

Block: In this filed the hash of the block, in which this transaction is aggregated the first time, gets stored. It is of type *[32]byte*.

MerkleRoot: Root of the Merkle tree to ensure integrity and the correct order for the transactions aggregated in this *AggTx*. It is a slice of type *[32]byte*.

These *AggTx* are added to the blocks similar to other transactions. They are listed in the $AggTxData$ slice.

Theoretical Aggregation Process

Funds transactions and aggregation transactions can be aggregated in two different ways. All other types of transactions are not taken into account. Furthermore, it is not possible to combine an already aggregated transaction aggregated by the sender and one by the receiver. This results in either the *From* or *To* slice to have a length of one.

1. Transactions can be aggregated by the sender. Then all transactions which are sent by a specific wallet are aggregated into one *AggTx*. When aggregating transactions this way, the *From* slice has a length of one, as there is only one sender included.
2. On the other hand transactions can also be aggregated by the receiver. Then all transactions sent to one specific wallet are aggregated into one *AggTx*. Here the *To* slice is only length one because all transactions are sent to one specific receiver.

When a miner combs through all open transactions he tries to aggregate as many open funds transactions as possible according to these two rules. If two or more transactions can be aggregated, their transaction hashes are written to the *AggTx's AggregatedTxSlice* and the transactions' boolean *Aggregated* will be set to *true*.

In the next step, the miner checks already closed blocks, whether there are transactions (either *FundsTx* or *AggTx*) which do match the chosen pattern (either aggregated by sender or receiver) and are not aggregated by now. If such historic transactions exist, they are also added to the *AggTx*. But they do not have an influence on the state during the post validation of a block anymore.

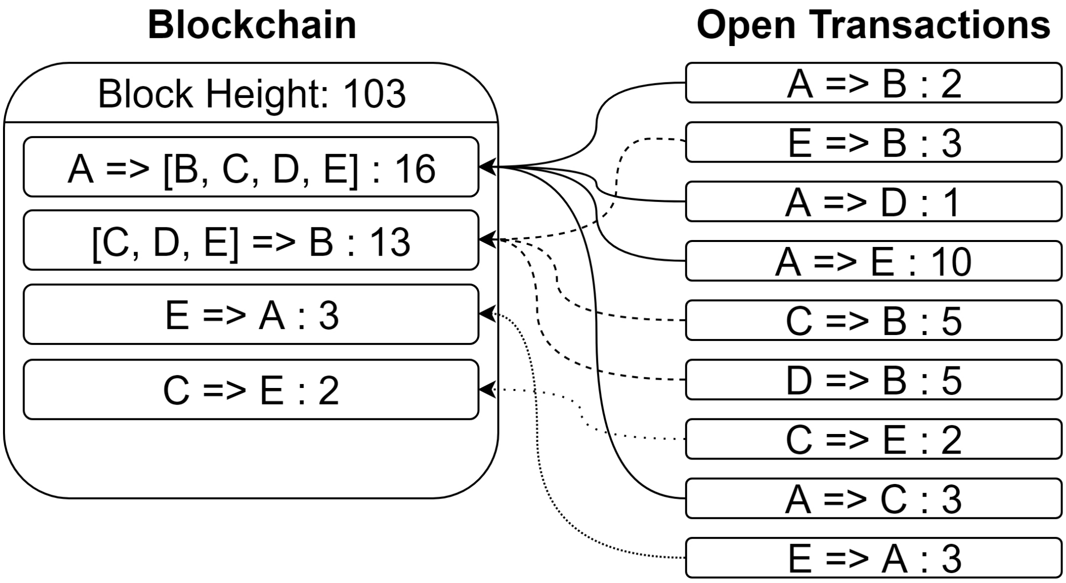


Figure 3.1: Concept transaction aggregation

In figure 3.1 the aggregation process is visualized schematically. The letters are wallets and the numbers are the amount of BAZO-coins sent (Similar to section 3.2.1). All open transactions are listed on the right side. In this example, the historic aggregation is omitted due to simplicity and only one of various possibilities is shown.

An algorithm (it is algorithm 1, which is further explained in subsection 4.2.1) does group these transactions in an optimal way, such that the fewest transactions are listed in the block, but the most transactions are validated. Without Transaction Aggregation, all these open transactions try to be in current block 103. When the block size is assumed to be limited to five transactions, with aggregation, there is still place for one more transaction whereas without transaction aggregation not even all nine transactions can be validated in the current block. Because BAZO only writes the hashes of transactions inside its blocks, this is possible. Consequently, with aggregation only the hashes of the two aggregated transactions and the two normal funds transactions, which cannot be aggregated in this block, are stored in the block's body. This is visible in figure 3.1.

Furthermore, it is also visible that it actually does not matter, how many transactions are aggregated in one *AggTx*. If *A* would have sent more transactions, still only one transaction is written into the block, but this transaction would aggregate more transactions. This is especially nice for a blockchain which is used in a case, where often multiple transactions are sent from one or to one peer. With small modifications, an imaginable example use case would be a blockchain which stores values sent from Internet-Of-Things devices always to the same receiver.

The miners do not earn a specific fee for aggregating. But they still receive all the fees which belong to the *FundsTx*. This results in miners that want to validate as many *FundsTx* as possible and thus earning as much as feasible. The more transactions they can aggregate the more transactions are in a block and they will get a higher reward. Since the block's size does not grow with every transaction, they can add more transaction into one block.

3.2.3 Double Linked Blockchain

The concept of a double linked blockchain is a result of the idea to remove all transactions from a block once all of them are aggregated in a later validated block. This is kind of a contradiction against the theory of a blockchain where all validated blocks are immutable. They are unchangeable because it would take too much effort to recalculate the complete chain since this adapted block, and additionally persuade over 50% of all miners to accept the newly created blocks.

In BAZO the block hash is calculated from various block related input fields. One of these variables is the *Merkle root*, which ensures transaction verification. It ascertains that transactions neither can be added to or removed from a block nor the ordering can be changed once a block hash is created [44]. Thus, the removing of transactions is only possible when a new additional block hash is calculated for every block because the old hash is becoming invalid, as soon as some transactions are removed. This new hash, called *HashWithoutTransactions*, is used always when the normal hash is becoming invalid. The normal hash becomes invalid because the *Merkle root* changes.

The goal and also the specification of the double linking is, that at least one link to the previous block is valid.

Block

Additionally to the common variables included in a block, the following fields are added in respect of the double linking of the blockchain:

Aggregated: It indicates if a block is aggregated and therefore does not contain any transactions anymore. It is of type *boolean*.

HashWithoutTransactions: This hash is used once all transactions from a specific block are aggregated. It can be calculated when not taking the transactions into account and as a consequence assuming an empty block. Thus, it is only possible to empty a block, once all transactions are aggregated and removed. It is of type *[32]byte*.

PrevHashWithoutTransactions: This field links the current block to the previous one once all transactions in the previous block are aggregated. It is of type *[32]byte*.

ConflictingBlockHashWithoutTx1: HashWithoutTransactions of the first conflicting block. It is of type *[32]byte*.

ConflictingBlockHashWithoutTx2: HashWithoutTransactions of the second conflicting block. It is of type *[32]byte*.

Theoretical Double Linking Process

In figure 3.2 the concept of a double linked blockchain is illustrated. Every block, expect the genesis block, can either be in the storage *Blocks With Tx* or *Blocks Without Tx*. This two versions of a block are indicated with *block-name*$_{w/}$ (including transactions) and *block-name*$_{w/o}$ (without transactions, meaning this block never contained transactions or all of them are aggregated by now). The increasing block number indicates which block is the ancestor, and the arrows point to them.

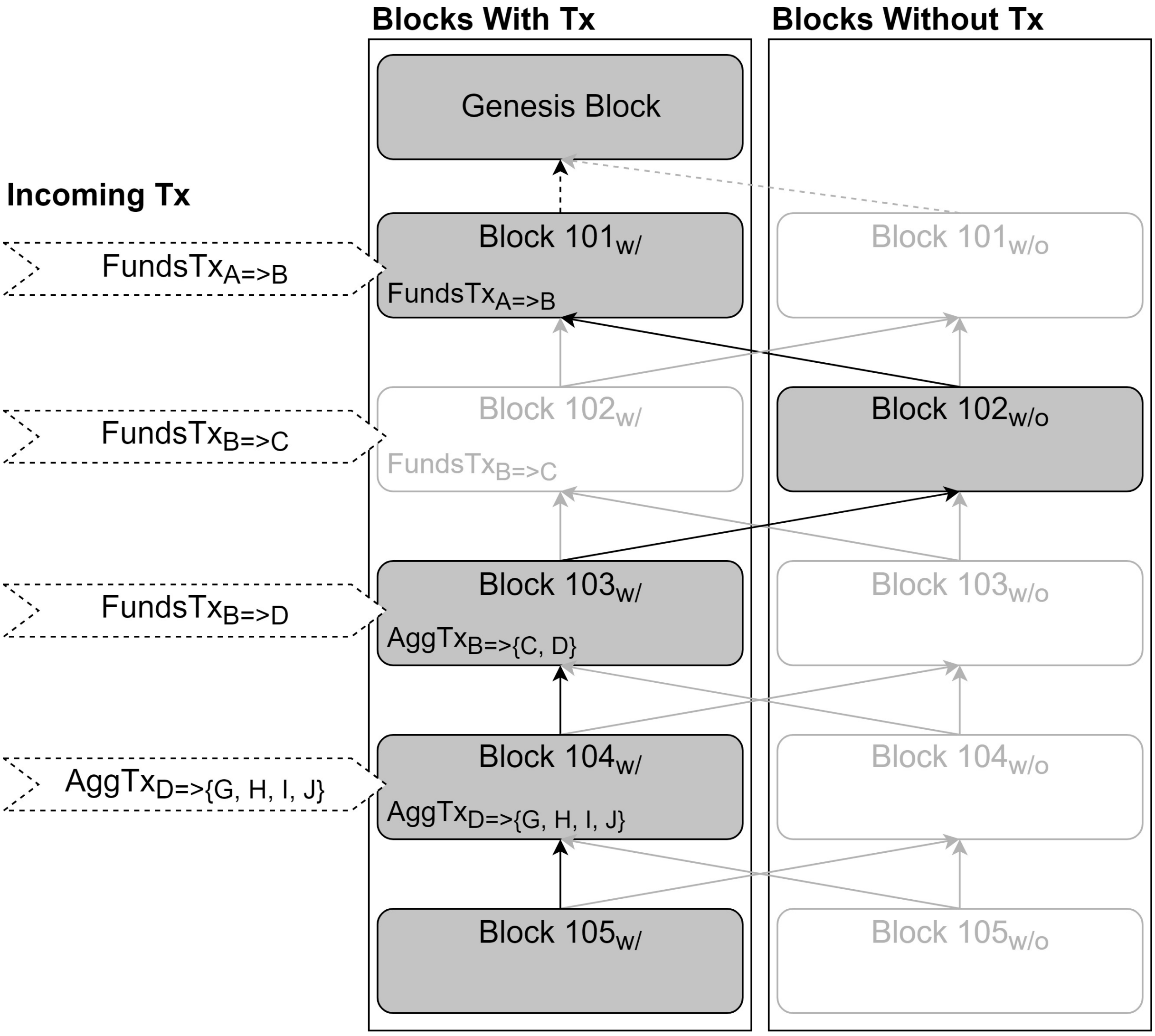


Figure 3.2: Concept double linked BAZO Blockchain

Every block, expect the genesis block, can contain transactions. These transactions are sent from clients to the network, what is indicated on the left side, as *incoming Tx*. Transactions are read as $FundsTx_{senderAddress=>receiverAddress}$ or when aggregated as $AggTx_{senderAddress=>\{receiverAddress\}}$. This happens as a historical aggregation in block 103 or while mining a new block denoted as the incoming *AggTx* also included in block 104.

As $FundsTx_{B=>D}$ reaches the network, the miners search in already validated blocks for other *FundsTx* or *AggTx* with either the same sender or receiver. In this example $FundsTx_{B=>C}$ in block 102 can be aggregated, which leads to the case where in block 102 all transactions are aggregated.

Once all transactions are aggregated and the block is out of the exclusion zone, it can be transferred to the storage without transactions. The exclusion zone is defined as the *current blockheight* minus *NO_EMPTYING_LENGTH* what ensures that the user-defined *NO_EMPTYING_LENGTH* last blocks are not moved even though all their transactions are aggregated. Meaning only blocks with a blockheight smaller than *currentBlockheight* - *NO_EMPTYING_LENGTH* are moved. Block 105 is not emptied yet despite the fact it does not contain any transactions. Once block 105 will be out of the exclusion zone, it will be transferred to the *Blocks Without Tx*. When a *NO_EMPTYING_LENGTH* of *2* is assumed, block 105 can be moved once a block with height 108 is appended to the chain.

When emptying block 102, it will be moved to the *Blocks Without Tx*, and the hash *HashWithoutTransactions* gets valid. Therefore block 103 is not linked to block 102 via the *Previous Hash* anymore but over the *PrevHashWithoutTransactions* now. It has to be ensured that one link between two consecutive blocks is always valid. In figure 3.2 this is indicated with the black arrows between blocks. This results in a valid chain indicated with grayish background color whereas all other blocks are only there for visual purposes and therefore slightly faded.

Chapter 4

Implementation

In this chapter, the implementation of the scalability improvement as well as the preparation for the test cases with the global network are discussed.

4.1 Performance Analysis

This section discusses the implementation and usage of the performance analysis part with the help of a global BAZO network.

4.1.1 Virtual Machines

For the performance analysis with the global network, virtual machines on *Amazon Web Services* (*AWS*), *Google Cloud Platform* (*GCP*) and locally at the University of Zurich (on server *b04*) were created. The locations of the virtual machines are visible in figure 4.1.

In *AWS*-datacenters the *t2.medium* machines with Ubuntu 18.04 are used. These virtual machines have 4GB of RAM [1].
On *GCP* alike specified machines are used. They are called *n1-standard-1*, run Ubuntu 18.0 and have 3.75GB of RAM [7].
The virtual machine located at the University of Zurich is running on the *b04* server. It uses Ubuntu 18.04 and has 2GB of usable RAM.

In the beginning, less powerful virtual machines were used. During testing the blockchain in the network, often an *out of memory* error occurred. Therefore, selecting the next bigger possible machine was necessary. On both, *GCP* and *AWS*, the next bigger ones with 1.7GB resp. 2.0GB, were still not powerful enough and therefore the next available configuration was selected. Luckily these were powerful enough and the error stopped appearing. On *b04*, the virtual machine was provided by the University and therefore no selection about the specs was possible. At the time of writing, the bootstrap miner runs at

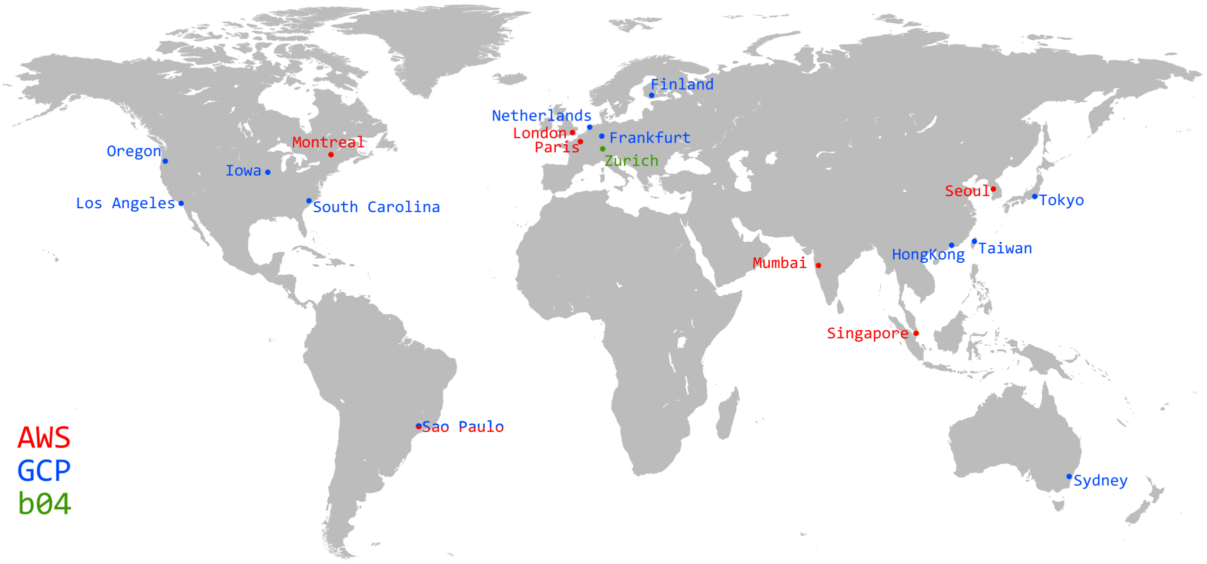


Figure 4.1: Global BAZO network. Image is created based on the world map from [26].

the University. Because the bootstrap miner operates as a root account, no transactions are sent from and to this wallet. Otherwise, new coins would be issued to the network. Due to the fact that no transactions are issued to the root miner permanently, there is no client running all the time. Thus, a machine with 2GB of RAM is powerful enough.

The client running on *b04* is, because of its root access, only used to set up the network and do configuration transactions. In all other regions, one client per virtual machine is set up as well. They send a fixed number of funds transactions to a predefined receiver. These transactions are sent by a simple bash *for*-loop with a *sleep* timeout in between.

To maximally ensure that no wallet is running out of funds, the funds need to be sent in a looping manner. The "route" is: *Oregon →South Carolina →Iowa →Sao Paulo (GCP) →Frankfurt →Finland →Netherlands →Sydney →Hong Kong →Tokyo →Taiwan →Los Angeles →London →Paris →Sao Paulo (AWS) →Seoul →Mumbai →Singapore →Montreal →Oregon.*

However, the direction of sending does not have an influence, because every transaction needs to be validated by ever miner anyways. Therefore, every transaction needs to be sent through the whole network. Furthermore, it can still happen that a miner runs out of funds, since the transaction fees also need to be paid or when one miner crashes.

4.1.2 Test Scenarios

In this section, the tree test scenarios, which are used for the evaluation of BAZO are described.

Different Block Size

In this test scenario, multiple test runs with different block sizes are executed. The block sizes tested are 1'000 byte, 5'000 byte and 20'000 byte. These sizes are chosen with respect to different circumstances. In Ethereum, the block size is currently a little below 20'000 byte [3]. Furthermore, because BAZO is planned to be an Internet-of-Things blockchain, a small block size, here 1'000 byte, is chosen. Thus it should be visible how many transactions per second BAZO can handle when the block size is kept small.

In these test runs, the block interval is set to 15 seconds and all 20 miners, defined in section 4.1.1, are used. The historic aggregation is disabled. The 19 non-root clients send 10'000 transactions each. For the test runs without transaction aggregation and a block size of 1'000 resp. 5'000 byte, the number of transactions sent is lowered. The lowering is needed because the $TPS_{calc.}$ is is very low and therefore it would take too long.

For the testing without transaction aggregation, nearly the same code is used. The only difference is how the transactions get added and how it is checked if a block is full already.

Different Block Interval

The test runs for different block intervals were designed similar to the ones for different block sizes. The only difference is that in these runs, the interval between the blocks is changing and not the block size. As for intervals, 15, 60 and 120 seconds are chosen and the blockchain was able to stabilize before the test cases started. The stabilizing is needed because the change of a block interval needs time to become consistent. The block size is fixed to 5'000 byte.

Again 20 miners were running in the network, and 19 clients are sending 10'000 transactions each. The number of sent transactions for large block intervals, with transaction aggregation disabled, is also reduced due to the same reason as in the previous chapter.

Blockchain Size

The testing of the blockchain's overall size (BCS) is done locally on a Windows 10 machine with 16GB of RAM. This is done because it is not important how fast transactions can be validated and how many miners are in a network. Therefore, the testing is done with four miners and a total of 15'000 transactions sent from three non-root clients. Between two transactions sent, a 0.1-second long break is inserted.

The block size is set to 5'000 byte and the interval between two consecutive blocks is 15 seconds.

In total three different scenarios are tested:

1. without transaction aggregation,
2. with transaction aggregation enabled,
3. with transaction aggregation and the emptying of blocks enabled as well.

In the 3. case, the *NO_EMPTYING_LENGTH* (described in section 4.2.2) is set to 10 blocks.

4.2 Transaction Aggregation

In this section the implementation of the transaction aggregation is described. The evaluation about it is in chapter 6.

4.2.1 Aggregation Of Transactions

Since only valid funds transaction and already validated *AggTx* can be aggregated, the aggregation takes place after a funds transaction is characterized as a valid transaction. Instead of adding this valid transaction directly into the block's body it gets added into a temporary slice. The transactions in this slice will be sorted by the sender's address and then by the transaction counter.

All different senders and receivers are stored into two maps. The number of occurrences in all open transactions, which can possibly be aggregated, are used as values. These maps are called *diffS* and *diffR* in this thesis.
This approach finds the best combination of how transactions will be aggregated (either by the sender or by receiver address). The `getMaxSenderReceiver(diffS, diffR)` function does return the sender and receiver with the most occurrences.

Algorithm 1 does group transactions in a way that open transactions either have the same sender or the same receiver. The output is then a new slice of transactions which matches the rules defined in subsection 3.2.2. It always takes the sender or receiver which occurs the most and groups its transactions together.
Those selected transactions are stored into the *txToAggregate* slice and aggregated with the function `AggregateTransactions(txToAggregate, b)` (on line 19 of algorithm 1). Then the algorithm removes this group of transactions from *possibleTxToAggregate* and recalculates the *diffS* and *diffR*. A miner repeats these steps until *possibleTxToAggregate* is empty. This ensures, that the highest maximal number of transactions is validated in a block.

The function `AggregateTransactions(txToAggregate, b)`does the aggregation. The aggregation of transactions already validated in previous blocks takes place during the process of building a new *AggTx*. After all grouped transactions, which were received by algorithm 1, are added to the temporary slice, BAZO searches transactions in closed blocks which do either have the same sender or the same receiver as the transactions in the temporary slice. These transactions can either be *FundsTx* or other *AggTx*. This is proceeded by an iteration over all closed blocks which still have transactions not aggregated. Once a matching transaction is found, it gets added to the temporary slice as well and the process continues with creating the *AggTx*. The *AggTx's* constructor does create the *AggregatedTxSlice* out of all matching transactions. After the new *AggTx* is created, it gets added to the block's *AggTxData* slice.

Algorithm 1: Split alid transactions for aggregation: splitSortedAggregatableTransactions *(block b)*

Input : block *b*

Output: Slice of transactions which can be aggregated and block *b*, into which they belong

```
get possibleTxToAggregate from TempList
get diffS and diffR
sort possibleTxToAggregate by senderAddress and then TxCnt
for moreTxToAggregate do
    maxSender, maxReceiver := getMaxSenderReceiver(diffS, diffR)
    if maxSender >= maxReceiver then
        forall tx in possibleTxToAggregate do
            if tx.From == maxSender then append tx to txToAggregate
            else keep tx in possibleTxToAggregate
        end
    else
        forall tx in possibleTxToAggregate do
            if tx.From == maxSender then append tx to txToAggregate
            else keep tx in possibleTxToAggregate
        end
    end
    remove txToAggregate from possibleTxToAggregate
    recount diffS & diffR
    AggregateTransactions(txToAggregate, b)
    set txToAggregate to nil
    if len(possibleTxToAggregate) == 0 then moreTxToAggregate = false
    else moreTxToAggregate = true
end
```

This implementation revealed the big problem with missing transactions which is described in section 5.4.2.

4.2.2 Double Linked Blockchain

For the concept of a double linked BAZO blockchain, the blocks were adapted as designed in section 3.2.3. Furthermore, a BoltDB bucket, namely *closedblockswithouttx*, for the blocks without transactions was created. This bucket holds all emptied blocks which do not include transactions anymore.

In every post validation step of a new mined or received block, it will be checked, if blocks can be emptied. This is done for all blocks, which are in the closed state but not aggregated yet. If a block can be emptied is determined by the transactions included in this block and its height. Only if all of the transactions are aggregated and the block is out of the

NO_EMPTYING_LENGTH, it can be moved to the newly created *closedblockswithouttx* BoltDB bucket.

Whenever a block can be emptied, the block's *FundsTxData* and *AggTxData* slices are set to *nil*, such that they do not contain any transactions. The transaction is still in the storage, but not linked to the block anymore.

Since the blocks moved to the *closedblockswithouttx* bucket do not contain any transactions anymore, rolling back these blocks can be problematic. Especially rolling back blocks with a height smaller than the current block's height minus *NO_EMPTYING_LENGTH* is difficult, because they do not know about the transactions anymore. When sharding would be implemented at this time, the *NO_EMPTYING_LENGTH* could have been the span between the current block's height and the *epoch block* (described in section 2.2.4). This would help insofar, that no forks, which already include emptied blocks, need to be resolved.

Because sharding for BAZO is not implemented yet, there are two possible solutions to minimize this risk. In the first one, the *NO_EMPTYING_LENGTH* is chosen that large, that there are no rollbacks including that many blocks for sure. The second one, also implemented, is letting all transactions know in which block they were validated for the first time. Thus, even an emptied block can be rolled back. However, this process is very computation intensive, because every transaction ever received need to be checked. This is caused by the fact that the used BoltDB is only a key-value database which does not allow search queries other than the key.

Chapter 5

Bug fixing

In this chapter, the major problems, found during the analysis of the BAZO blockchain with the help of a global network, are described. Additionally, the developed solutions and intentions behind it are indicated. This work was necessary because otherwise the improvement and analysis of BAZO would have been incorrect and useless because everything was built on top of a faulty working blockchain.

5.1 Forking

A fork is a blockchain typical phenomenon which can occur in three different types. The base behavior in all forks is the same. A fork splits one chain after a certain occasion into two or more different chains. Below the three types are listed, but only accidental forks are discussed in detail [16].

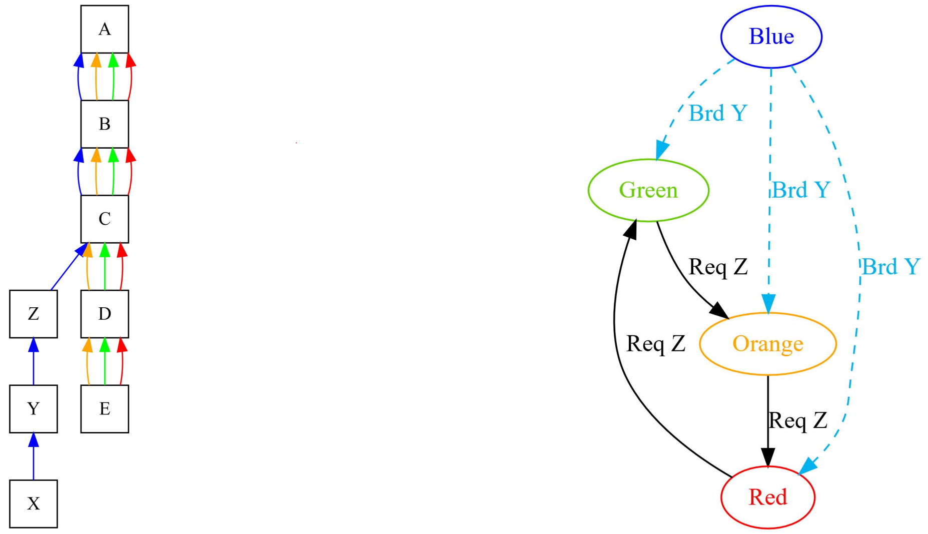
- *Hard Fork*: A hard fork is a permanent divergence in the chain. In the hard fork scenario nodes on one chain branch cannot validate blocks from the other chain because of changes in the protocol. Famous examples are the forking of Bitcoin where it forked to Bitcoin Cash and later also to Bitcoin Gold [32].
- *Soft Fork*: A soft fork, on the other hand, is a temporary divergence and therefore softer than a hard fork. It often occurs when not upgraded miners do not follow a new consensus rule for a certain time.
- *Accidental Fork*: Accidental forks appear most often of these three types. A fork of this type can be caused by two or more miners, with the same block height, mining and validating a new block at nearly the same time. Then the chain does quickly not know which branch is longer. When a new block is added to either of these two chain ends, it is clear which chain is the longer one. As soon as this happens, one branch will be rolled back and the miners concentrate on one chain again.
  Accidental forks also occur, when a miner does not receive blocks due to a connection outage.

5.1.1 Problem

The main problem BAZO had with forking was that the miners did sometimes not go back to one chain after an accidental fork. Under certain circumstances, no rollbacks were possible. The chain forked up to the state, where every active miner was working on his own chain.

Figure 5.1 illustrates the problem and its root cause based on a small example.
Graph 5.1a is read like this: Block A is the ancestor of block B and therefore visualized as $B \longrightarrow A$, because B includes the hash of A as previous hash. The different colored arrows are symbolizing different miners.
Figure 5.1b does only show the messaging, where block Y is broadcasted to the network with *Brd* Y and block Z is requested with *Req* Z. The arrows indicate the direction of the message. Again, the different colors are indicating different miners.

When the blockchain is at the point, where it is only mined up to D & Z, and these two blocks are mined at the same time, an accidental fork occurs. Here the miners do not know which fork branch is longer. The at this moment newest BAZO implementation rejects and deletes all blocks which were either invalid or belonged to a shorter or equally long chain. It is clear which chain is longer, as soon as the next block is mined, validated and broadcasted to the network.



(a) Forking blockchain

(b) Circling request

Figure 5.1: Description of forking problem and visualization of the root cause.

When E is the next mined and broadcasted block, the blue miner does make a block request for block D to the other miners and they all can give him the wanted answer. Once the blue miner received D, a rollback can be performed and the fork gets resolved. The blue miner has to request D, because it was discarded before, due to the fact, that at the time of receiving it, it did not belong to a longer chain.

When the next block is not D, but Y, the green, orange and red miner have to request block Z from the network, because of the same reason as in the previous paragraph.

When this request is done in a random manner, it is possible that they create a circling request (Figure 5.1b). They will never receive Z and therefore, the longer chain will not be accepted by the green, orange and red miner. The blue miner does not recognize that the others did not accept his mined and broadcasted block Y. Thus, the blue miner will start mining block X.
After the next broadcast, which is block X, the green, orange and red miner have to request blocks Y & Z form the network. The possibility, that a request is successful for one of those miners, is $(\frac{1}{3})^2$ and calculated by equation 5.1.

$$\textbf{Possibility} = \left(\frac{\#MinersWithBlock}{(\#Miners - 1)}\right)^{\#NumberOfBlocksToRequest} \tag{5.1}$$

Hence it is visible that the number of miners in the network ($\#Miner$) and the number of blocks to request ($\#NumberOfBlocksToRequest$) have a tremendous influence on the likeliness of a successful request.

A further problem was the fact, that the original BAZO implementation also discards all transactions when they cannot be added during the mining process of a new block. When the blockchain does fork accidental (*e.g.*, with blocks Z & D mined at the same time), the blockchain state is different on those chain branches until the miners focus on the longest chain again. Until this happens, some transactions may be valid in one fork branch and invalid in another one.

As a result, when validating blocks after a rollback, all transactions included in the blocks from the new branch have to be requested from the network. Here similar circling requests as described above can occur.

5.1.2 Developed Solution

The evolved solution for the forking issue can be divided into four parts: *broadcast block request*, *stashing of received blocks*, *better connectivity* and *transaction requests*.

To prevent the problem of circling requests in an easy manner the original block request to a random other miner is now a broadcast block request to the miners which are in the neighbor list. This does enhance the network traffic, but it provides a higher likeliness of receiving the requested block because more other miners are asked.

Enhancing the minimum number of miners in the neighbor list does enhance the likeliness of a successful broadcast block request. With a higher minimum, the miners do actively search for neighbors until this minimum number is reached. This provides better connectivity. Here also a dynamic timeout between two consecutive requests for neighbors makes sense. As a consequence, the miners now do more requests, when they only have a few neighbors and fewer requests when they are better connected.

Because the solutions listed above enhance the network traffic and the original BAZO implementation discards all blocks when they are not valid (*e.g.*, belonging to a shorter or equally long chain) a stash with received blocks is implemented. In this stash the last n received blocks, regardless of whether they are valid or not, are listed. This helps to lower the network traffic because before requesting a block from the network, a miner can search it in this stash. If it is in there, obviously no network request is needed.

The forking caused by the transaction requests can be handled in a similar manner. Thus, the request for a certain transaction is now also a broadcast request. Furthermore, especially to lower the network traffic, a stash for all, at a certain point, invalid transactions, is implemented. When a miner searches a specific transaction, as an example when the previously discarded transaction now is validated in a received block, it first can be searched in this stash. When the transaction is found there, it gets validated again and then processed as normal. If the transaction is neither in this stash nor in all other local storage facilities, a broadcast transaction request is executed.

It is important to mention that with these fixes, malicious behavior is not barred. When thinking about the example in figure 5.1a, where blocks Z & D were mined at the same time, a staking account can validate all transactions somehow including himself and thus, under certain circumstances, becoming a higher amount of coins in a shorter time than others. Then the likeliness of getting below the target in the PoS algorithm gets higher and with this, the possibility to be chosen to add the next block, and collect the reward, rises as well.

Investigation on this attack scenario and its prevention can be seen as future work because it busts this thesis goal and scope.

5.2 Block Size

BAZO does set a maximum block size and checks this twice. The first time when mining a new block for checking how many transactions can fit inside this block. The second time is when receiving another block from the network. Then it gets examined and validated again. When the block is too large at the second check, it is labeled as an invalid block and will not be added to the chain. Because no new blocks where added, the mempool never shrinks and in a new round the miners try to pack the same amount of transactions into the next block. This locks the network completely and neither transactions nor blocks get validated anymore.

This problem occurred only when lots of transactions were sent into the network.

5.2.1 Problem

The original BAZO version had a problem, where the check to prevent a block size overflow did not work properly. The block's size used at the first check was not increasing, because the transactions were truly added after the check. This resulted in a check on empty blocks all the time. This ended in too many transactions inserted into a new block.

During the second check in another miner, after receiving the new block, the block was, obviously, too large. Therefore, all miners sent blocks, which were too large into the network, and no new blocks where added to the chain.

5.2.2 Developed Solution

A simple counter does help to prevent the block size overflowing. Because the adding process is executed in a *for*-loop, a counter, which multiplies the number of already included transactions with the size of a transaction, does help. The downside is that this does only work if the part of a transaction written into a block is the same for all transactions. In BAZO this is the case because only the transaction hashes are included.

During this fix also a small but very effective scalability improvement was found. In the BAZO blockchain, only the transaction hashes are stored inside the blocks. These hashes have fixed sizes of 32 byte whereas the transactions have sizes between 83 (*ConfigTX*) and 362 byte (*StakeTX*). BAZO's original implementation did the block size check with those absolute transaction sizes. This resulted after the check was working properly, in far fewer transactions validated in a block than actually possible when executing these checks with the hash size.

5.3 Strange *Header.TypeID* & Connection Issues

In BAZO, every message sent through the network consists of a header and an optional payload. Whereas the payload is the actual message transferred, the header does contain the length of the payload as well as the TypeID. This TypeID does indicate the type of the message and peers distinguish how to handle an incoming message based on it. The header has a fixed size of 5 byte whereby the payload length takes 4 byte and the TypeID 1 byte. The TypeID is an 8 bit unsigned integer (*uint8*) what results in a value range from 0 to 255 and therefore in 256 distinct message types [44]. At the time of writing, 40 different message types are listed in the source code.

5.3.1 Problem

Despite the fact, that in BAZO only about 40 different message types are listed, extremely often received messages cannot be handled correctly because the TypeID is not known. Whenever an unknown message type is received, the connection to the sending peer gets aborted and needs to be reestablished in the next step. The received messages have often random looking TypeIDs in the range of an 8 bit unsigned integer.

When running multiple miners in the network, this leads to a constant reconnecting. Thus many messages sent through the network can only be sent delayed, or have to be requested by the receivers in a later step. This does, on one side, decelerates the network and thereby also the *TPS* and on the other side, increase the possibility of forks, because many transactions and / or blocks are not received or parsed correctly.

5.3.2 Developed Solution

At the time of writing, there is no fix for this issue. The implementation of the same checks before sending a message as when receiving a message, indicate that no unknown message types are sent to the network. All messages sent, pass the check for a valid TypeID but some of them do not pass the same test at the receiver's side. This does somehow suggest the assumption, that the concurrent receiving is the problem. When logging the start and end of *read* operations on one connection, they always alternate. This excludes simultaneously reading on one connection.

However, there are multiple ways to reduce the connection outage between miners. On one hand, a dynamic neighbor requesting interval does help in a way that miners do actively search other miners when they only know a couple of miners. At the time of writing, this requesting interval is between 5 sec, when less than two miners are known, and 30 seconds, when six or more miners are known. In between, it is dynamic according to the number of neighbors. This is also described in section 5.1.2. On the other hand and since most of the miners do not leave the network as soon as the connection is down, it is possible to directly try to reconnect to a miner. In BAZO all IP-addresses and ports of new miners are sent to the `checkHealthService()`Goroutine via the *iplistChan* channel. Thus reconnecting to a miner can simply be initiated by sending the IP-address and port again through this channel. Once it is received at `checkHealthService()`, a new connection will be reestablished.

5.4 Missing Transactions

In a peer-to-peer network, the network type which is used for blockchains, as well as in other network types it can happen, that a transaction gets lost. BAZO does work with a flooding type sending mechanism between different miners. Furthermore, in BAZO there are no acknowledgments sent between miners when transactions and / or blocks are well received.

BAZO works with a transaction counter (*txCnt*) for every wallet in the network. Furthermore, every miner does create the network's global state out of the received transactions. This means, that there is no message after a block which indicates that, *e.g.*, wallet *A* has now 20 coins. The miners calculate the balance for each wallet based only on the transactions they received.

5.4.1 Problem

Since BAZO miners do disconnect between each other very often, it happens regularly that not all transactions are received at all miners. In the original BAZO version, miners do not notice if transactions can be sent or not. This resulted in the issue, that at the moment a miner loses all connections to other miners, this miner still "sends" the transaction to the network. When no other miner is available, the sending miner nonetheless believed that the transaction was sent successfully, despite the fact, that this transaction never reached any other miner. There was no safety mechanism, which checks if there are connections to other miners and thus a connection to the network.

Since BAZO works with a transaction counter to prevent double spending and replay attacks, the next valid transaction is always the one with the transaction counter similar as in the miners' state [44]. When miner *A* receives transactions (form one wallet) with *txCnt* 10 up to 20, but the miners local state for this wallet is at *txCnt* 9, all these received transactions are not valid. And they will not be validated until the miner receives the transaction with *txCnt* 9.
This may never happen, because of the disconnection problem or it will take a lot of time until a block with this transaction is validated and sent to the network. When receiving this block, miner *A* may not know all transactions validated in there and thus, it can request the unknown ones. If the missing transaction with *txCnt* 9 is also requested cannot be known by the miner, because it only has unknown hashes which can be used for requests.

BAZO did not implement a mechanism where missing transactions can be requested based on the sender's address and *txCnt* which results in long validation times sometimes. This because every transaction had to be requested after the hash was received in a valid block.

5.4.2 Developed Solution

To completely solve this problem, more than one fix is needed. On one hand, a miner should recognize, when a connection is down, on the other hand miners should be able to request missing transactions based on the sender's address and the transaction counter. Hereby it is important, that only transactions are searched when a transaction with a higher transaction counter from the same sender is ready to get validated. This means, that only transactions between the current transaction counter from the local state and the transaction's *txCnt* are searched if there are transactions missing in between these two transaction counters.

As soon as a connection to another miner is down, the *peers.minerConns* map gets updated and the peer gets removed from the map. Even if there are no other miners in this map, from a sending-miner's perspective the transaction was sent to the network. Fixing this includes a new map, called *sendingMap*, into which all connections are written but do not get deleted, once the receiving-miner gets disconnected. Furthermore, all messages which cannot be sent immediately are added to a slice and are sent once the connection to the miner is reestablished. This map uses the IP-address and port of the other miners as the key because this is consistent even if miners reconnect to each other. The other characteristics (like *channel* or *connection*) of a BAZO peer do change with every reconnection. Therefore, it is not only checked, if a miner does exist in the map, but also if its values have changed and if they should be updated.

Once two miners reconnect to each other, all the messages from the slice are sent, such that they get the missed transactions and blocks. The slice which holds the not sent messages is limited, hence it will not influence the performance and prevents transaction starving.

Since BAZO uses the BoltDB, a simple GO specific key/value database, it is not possible to neither set secondary keys nor use specific queries to search transactions whenever the transaction hash (used as BoltDB key in the BAZO implementation) is unknown [33]. Therefore, when searching a transaction by the sender's address and *txCnt* either all transactions had to be checked, which is very bad in regards of performance, or a map with the *txCnt* as key needs to be implemented. With the second approach, an additional key/value mapping can be achieved, but the consistency needs to be handled by the user [11]. As a consequence, BAZO now has a map, called *TxcntToTxMap* with the *txCnt* as key and a slice of all transaction hashes matching this *txCnt* as value. When now a *SPECIALTX_REQ* with the sender's address and *txCnt* is received, the miners can look up transaction hashes in this map and do not have to iterate over every transaction ever received. This map is also used when searching locally before creating a transaction request.

Algorithm 2 shows how a miner determines if transactions are missing. It is not useful to only fetch the transactions missing between the state *txCnt* and the first transaction's counter, because other transactions may be missing as well. Thus the goal is an algorithm which finds missing transactions during the process of checking how many transactions can be validated in the block.

Therefore, a new map, called *missingTxCntSender* is implemented. This map takes the sender's address as key and sender's address, *txCnt* and a slice with missing transactions as value. A transaction is missing, if the transaction's *txCnt* is bigger than the local state's *txCnt* or the highest *txCnt* from the transactions which can be added at this point, plus one. Whenever this is the case, directly checked with the *for*-loop on line 6 of algorithm 2, the transaction counters between the currently applicable *txCnt* and the transaction's *txCnt* are appended to the missing transaction slice of this specific sender.

When missing transactions are detected in the previous step, these transactions are first searched in all local stores with the help of the previously described *TxcntToTxMap*. It can be the case that a miner does receive a transaction during the process of preparing a block, as an example when the miner reconnects to another one and this other miner now can send delayed messages, or if another miner broadcasts this transaction in the meantime. With this check, network requests can be decreased. When the transaction is not in the local storage, a *SPECIALTX_REQ* with the *txCnt* and the sender's address is sent to the network.

It is still possible that a transaction cannot be found, even with the *SPECIALTX_REQ*. Then, all transactions after this one cannot be added and this specific transaction will be searched with the next block again. This is designed that way, because when a request does not bring the desired transaction, the miner having this transaction may not be reachable at the moment. Then requesting immediately afterward does not help.

The method `checkBestCombination(allOpenTx)`, called on line 2 of algorithm 2, does return the maximal number of transactions which can be added with transaction aggregation.

Algorithm 2: Preparation of a new block: prepareBlock(block b)

```
Input  : block b
Output: none

...
openTx := checkBestCombination(allOpenTx)
forall tx in openTx do
    ...
    if missingTxCntSender[tx.From] == nil then create new map
    for i ← missingTxCntSender[tx.From].txCnt + 1 to tx.txCnt by 1 do
        append i to missingTxCntSender[tx.From].missingTransactions
    end
    append tx to openTxToAdd
end
...
forall sender in missingTxCntSender do
    limit length of sender.missingTansactions to int(blockInterval)
    forall tx in sender.missingTansactions do
        search tx local again
        if tx not found local then SPECIALTX_REQ
        if tx found then append tx to openTxToAdd
    end
end
...
sort openTxToAdd
forall tx in openTxToAdd do validate tx and add tx to TempList
splitSortedAggregatableTransactions(b)
```

All transactions which can be added, the *openTxToAdd*, are validated and written into a temporal storage, called *TempList*.

From this list, the algorithm splitSortedAggregatableTransactions() (described in 4.2.1) gets all transactions and aggregates them correctly.

Chapter 6

Evaluation

In this section, the results from the performance analysis are discussed. Therefore, the test cases defined in chapter 4.1.2 are used. There is a section about the block size, about the interval between blocks and about the blockchain's overall size. The chapter closes with the benefits and obstacles of transaction aggregation and a prospect about possible future work.

6.1 Different Block Sizes

Here, the evaluation and comparison between different block sizes and its influence on the *TPS*, with an initially defined amount of transactions sent to the network, is measured and listed. However, it is possible that not the same amount of transactions are sent to the network in each test run because of either the miner or the client crashes. This is mainly caused by an unrecoverable *SIGBUS* error. For certain test runs, the number of transactions sent is lowered on purpose, as described in chapter 4.1.2.

Table 6.1 does show the transactions per second rates for test cases with transaction aggregation enabled whereas table 6.2 shows the test cases without transaction aggregation. In both tables, the block size has unit *byte*, and the values belonging to transactions have unit *transaction* or *transactions per second.*

The actual block size (*ABS* and unit *byte*) is the available space in a block where transactions can be filled in. At the time of writing, it is $DefinedBlockSize - 658byte$. The *658 bytes* are used for block related values and therefore this space cannot be filled with transactions.

At a first look, it occurs strange that there are also TPS_{min} and TPS_{max} beside the normal *TPS*. This is due to the fact, that some miners need longer to fetch all transactions when they do not receive all transactions during broadcasts, as described in section 5.3. Also nearly always the virtual machines hosted on GCP had a lower *TPS* than the ones on AWS, what may be an indicator that they are slightly underpowered in regards of the random access memory. The TPS_{max} is an evidence of what speeds can be possible with transaction aggregation as it is implemented in this thesis. However, in chapter 6.2, it is visible that also with transaction aggregation, the TPS_{min} and TPS_{max} can be close to each other. Therefore it is strongly assumed, that this difference is caused by connection issues, which results in requesting more transactions.

Table 6.1: Table of different block sizes influencing the *TPS* with transaction aggregation enabled.

Block size		Transactions						
Defined	ABS	#sent	#validated	TPS_{sent}	TPS	TPS_{min}	TPS_{max}	$\frac{TPS}{TPS_{\text{calc.}}}$
1'000	342	179'328	178'196	33.1	28.0	14.6	32.7	39.2
5'000	4'342	181'933	181'933	33.3	28.5	20.9	33.2	3.1
20'000	19'324	181'613	181'613	33.0	28.0	16.7	32.9	0.7

Table 6.2: Table of different block sizes influencing the *TPS* with transaction aggregation disabled.

Block size		Transactions						
Defined	ABS	#sent	#validated	TPS_{sent}	TPS	TPS_{min}	TPS_{max}	$\frac{TPS}{TPS_{\text{calc.}}}$
1'000	342	19'000	19'000	36.3	0.6	0.6	0.6	0.7
5'000	4'342	74'960	74'834	34.7	7.0	7.0	7.0	0.8
20'000	19'342	181'078	181'078	33.5	27.3	27.2	27.4	0.7

As visible in table 6.1, the block size does not take a big influence on the reached *TPS* value. This does have multiple reasons:

1. **Transaction Aggregation:** Since transactions are aggregated, not every transaction needs space in a block, and thus, more transactions fit into one block. Actually it does not matter, if there are two transactions from *A* to *B* or if there are one hundred transactions from *A* to *B*. With transaction aggregation, only one transacting will be written into the block. Consequently, it is good to group as many transactions as possible and bad, not to group any transaction, because it needs too much space in comparison.

2. **Unlimited *AggTx* Size:** If a transaction does not fit in one block, it is likely that it is validated in the next block because there is no limit in how many transactions can be written into one *AggTx* and because of the *splitAndSort*-algorithm's design (explained in section 4.2.1, algorithm 1). This algorithm takes the senders or receivers which have most awaiting transactions to be validated. Thus, the more transactions from one sender or to one receiver are in the mempool, the more likely it is that they get validated. Because the transactions from / to one wallet, which cannot be validated in a block, are still in the mempool, it is likely that there are more transactions matching the selection criteria for the next block. Since the size of an *AggTx* is unlimited, they are able to aggregate as many transactions as possible.

3. **Test Case:** In the test cases, there are maximal 20 users in the network. Thus it is likely, that the same sender or receiver is found quickly. This does help to keep the *TPS* at a high level. It is obvious that transaction aggregation works better the more similar sender or receivers are in the transactions. But since BAZO is planned to become an IoT blockchain, where multiple IoT devices send their data to one receiver, this limited number of wallets is not a problem.

4. **Transaction Sending:** Transactions are sent approximately every $^1/_2$ second. If transactions would only be sent after the previous transaction is validated in the network, aggregating by sender would not work. The current acknowledgment a miner sends to a client is only confirming that a certain transaction is sent to the network. A client does not know if and when the sent transaction is validated. Thus, aggregating by sender is possible.

The $\frac{TPS}{TPS_{\text{calc.}}}$ does indicate by which factor the aggregation increases the maximum possible $TPS_{calc.}$ value. Thus with a block size of 1'000 byte, the version with transaction aggregation can theoretically handle roughly 39 times more transactions than without aggregation. It is clear, that with a smaller $TPS_{calc.}$ and a constant *TPS*, this factor increases.
The $\frac{TPS}{TPS_{\text{calc.}}}$ for a block size of 20'000 byte is below one, because, theoretically, over 600 transactions fit into a 20'000 byte block what results in a $TPS_{calc.}$ of about 40 transactions. This $TPS_{calc.}$ is higher then the TPS_{sent} and therefore, the ratio cannot be bigger than one.
Summarizing this, it is not possible to generally say, that with transaction aggregation it is exactly 39.2 respectively, 3.1 times faster, because this numbers are highly influenced by the actual *TPS* which can not be higher than the TPS_{sent}. Thus, sending more transactions probably increases the *TPS*, and therefore also the $\frac{TPS}{TPS_{\text{calc.}}}$ ratio, even more.

In table 6.2, the fraction $\frac{TPS}{TPS_{\text{calc.}}}$ can be understood as degree of utilization in terms of defined and actual block size.

Figure 6.1 does visualize the output of tables 6.1 and 6.2. The scalability improvement is clearly visible. One can say, that with transaction aggregation, the block size is not a limiting factor anymore if there are either same senders or same receivers in the transactions. This is visible in figure 6.1, because the blue bars are nearly constant, whereas the orange ones, standing for test runs without aggregation, are increasing with bigger block sizes. The grey bars do visualize the $TPS_{calc.}$.
Furthermore it is surprising, that the orange bar is not higher in the test run with a block size of 20'000 byte. The block's size is not limiting the *TPS* anymore, since the $TPS_{calc.}$ is higher as the TPS_{sent}. This may indicate, that the connection issues described in section 5.3 and 5.4 are limiting the network throughput. Unfortunately, this assumption can neither be confirmed nor rejected because there is no fix for this problem yet.
The TPS_{sent} is indicated by the green line.

The shrinking of the TPS_{sent} in figure 6.1 is not related to the block size, as it may could be assumed. Sending the transactions, even with a fixed interval, is still not always equally fast. It is related to the timespan in which a client receives an acknowledgment from a

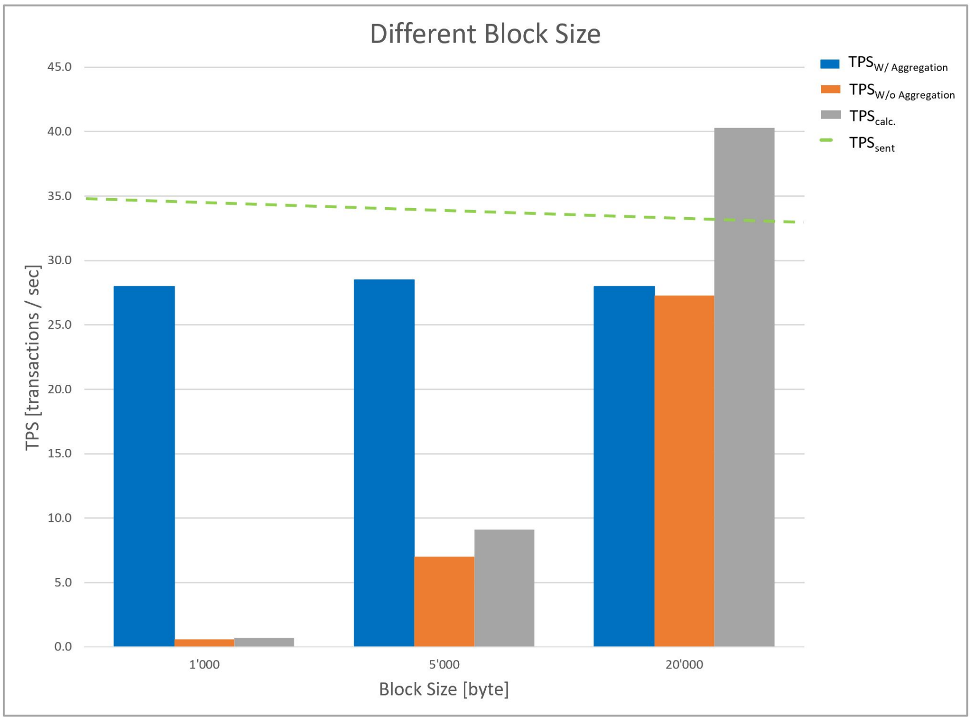


Figure 6.1: Different block sizes and its influence on the *TPS* with and without aggregation

miner after sending the transaction. This timespan can slightly differ, and thus, it can have a bigger influence when sending a lot of transactions.

In the test runs with a block size of 1'000 byte, the version with transaction aggregation can handle that much more transactions because about 10 transactions fit into one block. These 10 transactions can be aggregation transactions aggregating way more transactions and therefore increasing the *TPS*. Here point two from the listing above does have an influence because 19 different wallets are in the network. When aggregating these transactions perfectly, it would result in 19 *AggTx* transactions. This would overflow the block size by nine transactions. If transactions from one sender cannot be validated in block n, there will be all these transactions plus the newly received ones for block $n + 1$. Thus, during the preparation of block $n + 1$, this specific sender will have more transactions than a sender whose transactions get validated in block n and therefore all its transactions get validated then.

The global distance of the network is not influencing the performance a lot since the ping-latency between the used virtual machines was usually below half a second during the test runs.

The block size is not limiting the BAZO version with transaction aggregation up to the point, where only distinct senders and receivers are sending and receiving the transactions. But since BAZO is going to be an Internet-of-Things blockchain, where different IoT-devices often send data to the same receiver, this looks quite unreachable. Therefore, in regards to the block size, transaction aggregation does increase the *TPS* especially for small blocks and thus it looks very promising.

6.2 Different Block Intervals

Here the comparison and evaluation between different block intervals, with a given number of transactions sent to the network, is measured and listed.

Table 6.3 does show the *TPS* rates for test cases with transaction aggregation enabled whereas table 6.4 shows the test cases without transaction aggregation. In both tables, the block interval has unit *seconds*, and the values belonging to transactions have unit *transactions* or *transactions per second.*
In table 6.3 the fraction $\frac{TPS}{TPS_{\text{calc.}}}$ can be seen as improvement factor in contrast to the theoretical maximum and in table 6.4 as degree of utilization.

The *ABI* (actual block interval) is an indicator if the blockchain does validate blocks in the user defined interval. It has unit *seconds* and is the average timespan between two blocks. Since the validation speed is set with the help of the *target*, it is not exactly the set interval [19]. This *target* is adapted every n blocks, whereas n is user defined.

Table 6.3: Table of different block intervals influencing the *TPS* with transaction aggregation enabled.

Block interval		Transactions						
Defined	ABI	#sent	#validated	TPS_{sent}	TPS	TPS_{min}	TPS_{max}	$\frac{TPS}{TPS_{\text{calc.}}}$
15	18.8	181'933	181'933	33.3	28.5	20.9	33.2	3.1
60	66.8	190'000	190'000	34.8	34.4	33.9	34.7	15.3
120	118.4	181'278	181'278	33.8	32.9	30.2	33.4	28.5

Table 6.4: Table of different block intervals influencing the *TPS* with transaction aggregation disabled.

Block interval		Transactions						
Defined	ABI	#sent	#validated	TPS_{sent}	TPS	TPS_{min}	TPS_{max}	$\frac{TPS}{TPS_{\text{calc.}}}$
15	14.3	74'960	74'834	34.7	7.0	7.0	7.0	0.8
60	63.5	78'375	78'375	36.3	2.0	2.0	2.0	0.9
120	111.4	18'000	18'000	34.4	1.1	1.1	1.1	1

The test runs with different block intervals look similar to the test runs with various block sizes. The *TPS* can be increased when transaction aggregation is used. Especially because the block interval and the *TPS* without aggregation are behaving inversely proportional, these test runs show the advantages nicely. The relationship between the *TPS* and the block interval is inversely proportional because, with a higher block interval, fewer blocks can be validated, and thus, less transaction as well.
The block size, on the other side, is related proportional to the *TPS*, because bigger blocks can handle more transactions. This results in a higher *TPS* value.
Consequently, the version without aggregation can theoretically handle the most transactions with a tiny block interval and huge blocks.

Similar to the different block sizes, the block interval is not a *TPS* limiting factor anymore. Furthermore, transaction aggregation also allows validating more transactions per second, as already stated before.

During one test run, with a block interval of 60 seconds, the *TPS* is close to the TPS_{sent}. In this test run, due to certain unknown circumstances, the TypeID issue (section 5.3) was not influencing so much as during other test runs.

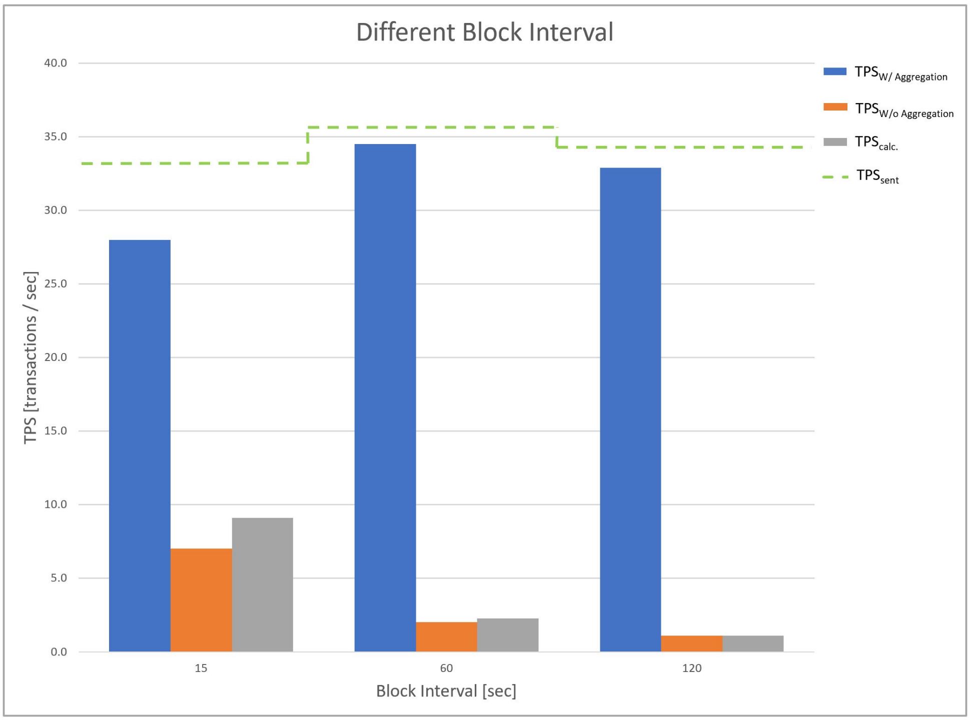


Figure 6.2: Different block intervals and its influence on the *TPS* with and without aggregation

In figure 6.2 the differences of the TPS_{sent}, which is indicated with the green line, were relatively big and thus it has different heights. The differences are again caused by small differences between sending a transaction and receiving the acknowledgment from a miner.

The difference between the *TPS* with aggregation (blue bars in cart 6.2) and without (orange bars) is even bigger here, since the block interval and the *TPS* are inversely proportional.

It is not appropriate, to say, that with a higher block interval, the *TPS* can be increased for BAZO with aggregation enabled. Theoretically, the different block intervals should not have an effect on the *TPS* up to a certain number of different senders and or receivers. However, the differences can be founded on the TypeID issue again. This, because with the two higher block intervals, four to eight times fewer blocks have to be sent through the network compared to the 15 seconds interval. Therefore, the miners probably disconnect less often and they have more time to fetch transactions or blocks, which they never received.

Similar to the test with different sizes of blocks, the global distribution of miners and clients did not have a huge influence. The latency for a ping-test was usually around half a second.

Summarizing these outcomes results in the same perception as in section 6.1. Transaction aggregation does definitely increase the transactions possible to handle per second. Especially for a large block interval, similar to small defined block sizes, summarizing transactions increases the throughput enormously. Since the relation between the block interval and the *TPS* is inversely proportional, the difference in regards to the *TPS* with and without aggregation is even bigger, as with different block sizes.

6.3 Blockchain's Overall Size

In this section, the blockchain's overall size is analyzed and the findings, based on the test scenario defined in subchapter 4.1.2, are discussed.

As it is visible in graph 6.3, the blockchain size can be reduced with aggregation and even more with aggregation and emptying of blocks, once all transactions are aggregated. The difference between only aggregating and aggregating with emptying is not extraordinary big, because, in this test case only three different transactions are written into a block with aggregation. When emptying a block, the block's size gets reduced only around three times the size of a transaction hash. The more different transactions are listed in a block, the bigger this difference will be.

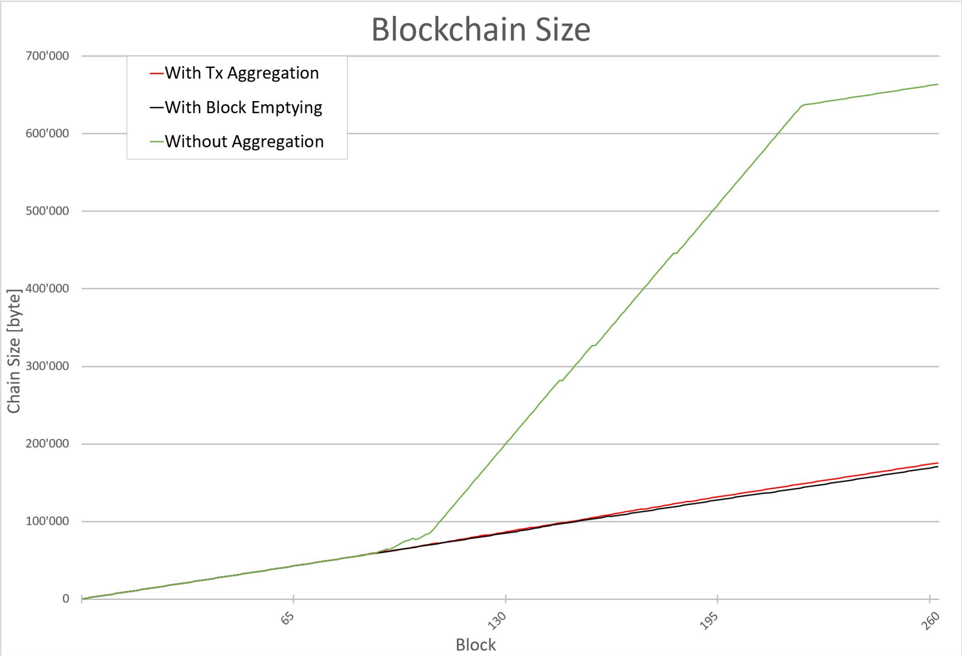


Figure 6.3: Differences regarding the size of the blockchain with aggregation, with aggregation and emptying of blocks and without aggregation.

When a BAZO version with transaction aggregation (red or black line in figure 6.3) is compared to the version without (green line), a huge difference is visible. The difference is this big, because with transactions aggregation enabled, around three transactions are validated in each block, whereas without aggregation, roughly 135 transactions get aggregated in each block. These 135 transactions are also the maximum capacity of a block with the defined size of 5'000 byte. The two major kinks are caused by the start and end of sending transactions, whereas the smaller ones are caused by rollbacks. This graph shows the possibility of having a smaller overall blockchain size with transaction aggregation and emptying of blocks.

However, since self-contained proofs are not implemented in this aggregation technique, a new joining miner still has to fetch all transactions, no matter if BAZO is aggregating or not. And because aggregating transactions brings an extra transaction each time some transactions get aggregated, a new joining miner has to fetch even more transactions actually. This will change, once self-contained proofs are implemented. Then only the transactions since, *e.g.*, the last *epoch block* needs to be fetched from the network. This reduces the amount of data which needs to be fetched.

6.4 Benefits Of Transaction Aggregation

Transaction aggregation does definitely allow more transactions in one block. Thus the overall throughput increases and at the same time the block size and the overall chain size can be kept small. Furthermore, the block interval can be enlarged, which reduces network traffic. However, transaction aggregation does help the most, if there are similar senders or receivers of transactions. It performs better, the more transactions with the same sender or receiver are sent. As BAZO is becoming an IoT blockchain, where many IoT nodes send their transactions to one receiver, this aggregation approach helps in scaling the blockchain.

As stated above, the implementation presented in this thesis does not help if there is no overlapping in terms of sender or receiver. Therefore, another aggregation technique should be used. A different possibility would be: Aggregate transactions *(A →B : 5)* and *(B →A : 20)* as one transaction *(B →A : 15)*. Here the only the final amount and the direction of the transaction gets written into the blockchain. This idea is kind of similar to *state-channels* proposed in section 2.2.6.

Although transaction aggregation already works well here, its full power may only be visible once sharding and self-contained profs are implemented and combined with this technique.

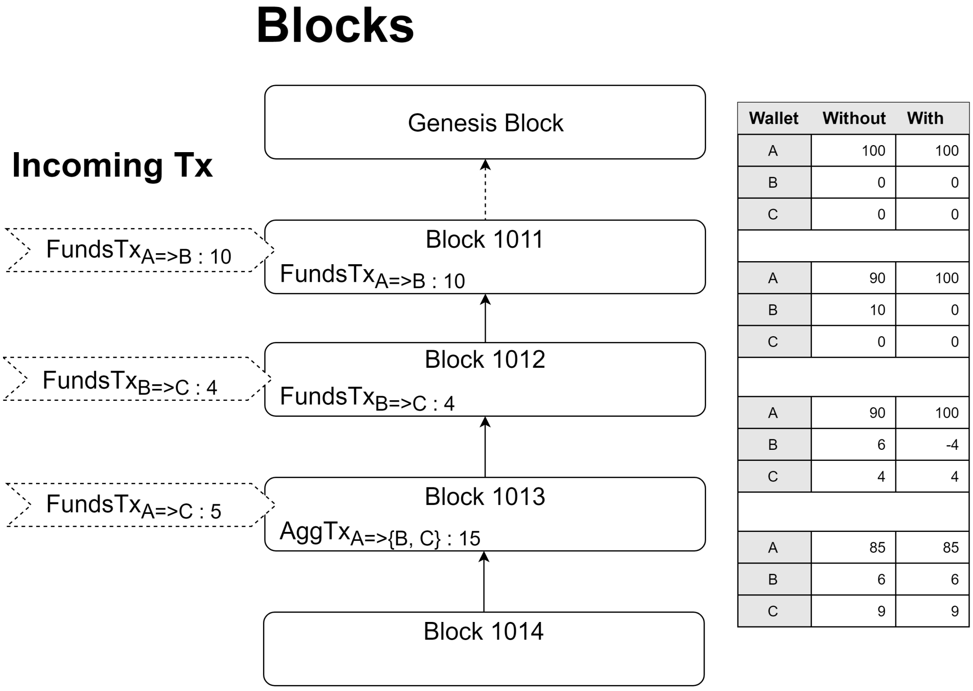
6.5 Obstacles Of Transaction Aggregation

The intention behind double linking the blockchain is emptying all blocks once they are secure enough. A block is secure enough when it is accepted by the majority of miners, and thus, will not be included in rollbacks anymore. The emptying helps to save storage, as visible in section 6.3. However, the emptying of validated blocks is kind of a contradiction against the core concept of a blockchain, where secure blocks are immutable and cannot be changed again. Thus, some impediments occur, especially when a new miner joins the network and wants to start mining.

6.5.1 Join As A New Miner & Order Of Transactions

When aggregating transactions in a historic manner, as it is described in section 3.2.2, various problems and difficulties can occur. They are described with the help of figure 6.4.

In figure 6.4 three FundsTx are incoming to the blockchain and get validated in in blocks *1011*, *1012* and *1013*. As it is visible, the third transaction ($FundsTx_{A=>C\,:\,5}$) can be aggregated with the first transaction ($FundsTx_{A=>B\,:\,10}$), because of the similar sender. This results in the $AggTx_{A=>\{B,\ C\}\,:\,15}$ in block *1013*, and the removing of $FundsTx_{A=>B\,:\,10}$ in block *1011*.
The table on the right side shows the balances for the three wallets A, B and C with and without aggregation, before, between and after the three blocks are validated.



Wallet	Without	With
A	100	100
B	0	0
C	0	0
A	90	100
B	10	0
C	0	0
A	90	100
B	6	-4
C	4	4
A	85	85
B	6	6
C	9	9

Figure 6.4: Transaction aggregation and the balance.

As it is now visible in the table, the balances for A and B are not the same when aggregating the transaction as when not aggregating them. This can lead to problems, especially when restarting or joining the network after transactions are already sent. Since BAZO fetches all blocks from the last validated one to the genesis block first and afterward validates them in the correct order, moving and aggregating transactions is problematic.

As example, when the transactions $FundsTx_{A=>B\,:\,10}$ and $FundsTx_{A=>C\,:\,5}$ get aggregated to $AggTx_{A=>\{B,\ C\}\,:\,15}$ and thus transaction $FundsTx_{A=>B\,:\,10}$ moves from block *1011* to

1013, *B* does not have enough funds for transaction $FundsTx_{B=>C\,:\,4}$ at the point of validating block *1012*. This is visible in the middle two sub-tables where the balance of *B* is not the same with and without aggregation. When a new miner is joining the network, which uses transaction aggregation, and the $FundsTx_{A=>B\,:\,10}$ is not in Block *1011* but in *1013*, this new joining miner is not able to validate block *1012*, because *B* does not have enough funds.

One Way of eluding this is a credit-like behavior on startup. This concept allows a wallet to have a negative balance during the startup process. At the end, similar to the fourth small table in figure 6.4, the balances with aggregation enabled are the same as when validating each transaction without aggregation. As long as all transactions are validated, the order of validation does not play a substantial role. At a participation of a miner, this new miner only validates transactions which are already validated from other miners in the network. If the bootstrap miner tries to send invalid transactions to the new miner, the currently joining miner will find them, either by invalid block hashes, or when other miners are refusing its mined blocks later. Consequently, with this credit like behavior during the participation, it is possible to validate block *1012* even with aggregation and join the network.

Furthermore, once sharding is implemented, it can be assumed, that new miners are only able to join when an *epoch block* is inserted to the chain. This is needed because of the load balancing and the division into shards. Because these *epoch block*s are similar for every miner, and thus every miner agrees on all blocks before this *epoch block*, joining is not problematic, even if the transactions are not validated in the correct order. Once self-contained proofs are merged and therefore not all transactions are needed to prove that a wallet has enough funds, the transactions before an *epoch block* will not be used anymore.

6.5.2 Join As A New Miner & Nonce

Joining as a new miner, when blocks are already emptied, is problematic since the nonce of a block is calculated with the help of the wallets' balances. Here, similar problems as in the previous subsection can occur, and blocks are not validated because the nonce is incorrect at this point. This should also be possible to prevent, when not checking the nonce on startup. It is also possible to argue, that these blocks are validated in the network already and therefore secure.

This problem probably will also shrink, when sharding is merged, because then it is ensured, that at the time an *epoch block* is inserted, the network agreed on one block. Thus all miners approve blocks up to this *epoch block* as valid. When only emptying blocks up to this *epoch block*, only commonly accepted blocks are emptied and the nonce of these blocks does not have to be checked during startup.

6.6 Future Work

Future work on BAZO should definitely include a fix for the problem with the strange TypeIDs for messages sent in the BAZO network, described in section 5.3. This should have the highest priority because it will reveal, how powerful sharding and transaction aggregation is with stable connections between the miners.

In a possible next step, sharding, transaction aggregation, self-contained proofs and other IoT adaptions should be merged into one project, such that the combined power can be used. Only if all these mechanisms are combined, the true power and speed of BAZO can be revealed.

Nice and very useful would also be, if a user does not have to take care of the transaction counter in the future. Probably there is a way, how the transaction counter can be fetched from the state for every incoming transaction. Thus, the transaction counter is out of the user's view and usage of the blockchain will become easier.

Chapter 7

Summary and Conclusions

As seen in the previous chapters, the implemented transaction aggregation does work in a global network of up to 20 miners. The development process was hampered due to the fact that many defects and issues were only found once the implementation was tested with multiple miners. Fixing these problems from the original BAZO implementation took a tremendous amount of time. Because of this and the fact that some old issues are still not fixed, sometimes the result in regards to the *TPS* may be not as unambiguous as they can be. However, transaction aggregation, as it is implemented here, does truly help to increase the maximal number of transactions per second. This is especially visible when the block size is chosen small, or when the block interval is big.

Furthermore, it is also proven, that the overall blockchain size can be reduced with the techniques presented in this paper. Also here, the full power of it will be visible once sharding and self-contained proofs are merged.

Bibliography

[1] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 29.03.2019.

[2] Arrays, Slices (And Strings): The Mechanics Of 'Append'. https://blog.golang.org/slices. Accessed: 04.03.2019.

[3] Ethereum Average Blocksize Chart. https://etherscan.io/chart/blocksize. Accessed: 04.02.2019.

[4] Git Large File Storage. https://git-lfs.github.com/. Accessed: 06.04.2019.

[5] ./jq. https://stedolan.github.io/jq/. Accessed: 06.04.2019.

[6] Lubuntu. https://lubuntu.net/. Accessed: 15.04.2019.

[7] Machine Types. https://cloud.google.com/compute/docs/machine-types. Accessed: 29.03.2019.

[8] Mehr Informationen Über VisaNet. https://www.visaeurope.ch/de_CH/uber-visa/visanet.html#2. Accessed: 02.02.2019.

[9] Performance. http://redbellyblockchain.io/Benchmark.html. Accessed: 02.02.2019.

[10] Segmentation Fault (SIGSEGV) Vs. Bus Error (SIGBUS). https://www.geeksforgeeks.org/segmentation-fault-sigsegv-vs-bus-error-sigbus/. Accessed: 07.04.2019.

[11] Using Two Separate Keys In BoltDB. https://stackoverflow.com/questions/36111698/using-two-separate-keys-in-boltdb/36115609#36115609. Accessed: 10.04.2019.

[12] Visa Acceptance For Retailers. https://usa.visa.com/run-your-business/small-business-tools/retail.html. Accessed: 02.02.2019.

[13] VISA Fact Sheet. https://www.visaeurope.ch/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf. Accessed: 02.02.2019.

[14] Was Sind Altcoins? https://www.coinpro.ch/was-sind-altcoins/. Accessed: 02.04.2019.

[15] Zilliqa Is A Scalable, Secure Public Blockchain Platform. https://zilliqa.com/about-us.html#general. Accessed: 03.02.2019.

[16] Andreas M. Antonopoulos. *Mastering Bitcoin, 2nd Edition.* O'Reilly Media, 2017.

[17] Collis Aventinus. Age Of Scale: How Can Blockchain Systems Become Powerful Enough For A Global Audience? https://cointelegraph.com/news/age-of-scale-how-can-blockchain-systems-become-powerful-enough-for-a-global-audience. Accessed: 03.04.2019.

[18] Matt B. To Reduce Or Not To Reduce The Block Size? https://medium.com/chainrift-research/to-reduce-or-not-to-reduce-the-block-size-ed42795f3891. Accessed: 09.04.2019.

[19] Simon Bachmann. Proof Of Stake For Bazo. https://files.ifi.uzh.ch/CSG/staff/bocek/extern/theses/BA-Simon-Bachmann.pdf. Accessed: 02.02.2019.

[20] Paddy Baker. Zilliqa CEO: Mainnet Launch Shows Sharding Works. https://cryptobriefing.com/zilliqa-mainnet-launch-sharding/. Accessed: 17.04.2019.

[21] Shehar Bano, Mustafa Al-Bassam, and George Danezis. The Road To Scalable Blockchain Designs. *USENIX; login: magazine*, 2017.

[22] Matthew Beedham. Blockchain Sharding Made So Simple Your Dog Would Understand. https://thenextweb.com/hardfork/2019/01/18/explainer-blockchain-sharding-beginners/. Accessed: 09.04.2019.

[23] Roman Blum. Scalability For The Bazo Blockchain With Sharding. https://github.com/rmnblm/papers/blob/master/sharding_concept/main.pdf. Accessed: 02.02.2019.

[24] Deb Cobb. Best Practices In Blockchain Testing. https://www.neotys.com/blog/best-practices-blockchain-testing/. Accessed: 02.02.2019.

[25] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating The Red Belly Blockchain. *CoRR*, abs/1812.11747, 2018.

[26] Crates. World Map Blank Without Borders. https://de.wikipedia.org/wiki/Datei:World_map_blank_without_borders.svg. Accessed: 05.02.2019.

[27] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On Scaling Decentralized Blockchains. In *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[28] Brian Curran. What Is Proof Of Authority Consensus? Staking Your Identity On The Blockchain. https://blockonomi.com/proof-of-authority/. Accessed: 03.02.2019.

[29] Aniket Dongre David Schatsky, Amanpreet Arora. Blockchain And The Five Vectors Of Progress. https://www2.deloitte.com/insights/us/en/focus/signals-for-strategists/value-of-blockchain-applications-interoperability.html. Accessed: 09.04.2019.

[30] Aziz Dolce. Blockchain Scalability Solutions: Overview Of Crypto Scaling Solutions. https://masterthecrypto.com/blockchain-scalability-solutions-crypto-scaling-solutions/. Accessed: 03.02.2019.

[31] EdChain. Blockchain FAQ #3: What Is Sharding In The Blockchain? https://medium.com/edchain/what-is-sharding-in-blockchain-8afd9ed4cff0. Accessed: 22.03.2019.

[32] Werner Grundlehner. Der Bitcoin Eilt Zum Nächsten Rekord – Aber Abspaltungen Verunsichern Investoren. https://www.nzz.ch/finanzen/bitcoin-rekord-aber-abspaltungen-verunsichern-die-investoren-ld.1325385. Accessed: 24.01.2019.

[33] Ben Johnson. An Embedded Key/Value Database For Go. https://github.com/boltdb/bolt. Accessed: 17.03.2019.

[34] Kenny Li. The Blockchain Scalability Problem & The Race For Visa-Like Transaction Speed. https://hackernoon.com/the-blockchain-scalability-problem-the-race-for-visa-like-transaction-speed-5cce48f9d44. Accessed: 09.04.2019.

[35] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A Secure Sharding Protocol For Open Blockchains. In *Proceedings Of The 2016 ACM SIGSAC Conference On Computer And Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[36] Mark McGranaghan. Go by Example: Channels. https://gobyexample.com/channels. Accessed: 02.04.2019.

[37] Mark McGranaghan. Go By Example: Goroutines. https://gobyexample.com/goroutines. Accessed: 02.04.2019.

[38] Noisefloor. Tmux. https://wiki.ubuntuusers.de/tmux/. Accessed: 06.04.2019.

[39] Eric Olszewski. State Channels For Dummies: Part 1. https://medium.com/blockchannel/counterfactual-for-dummies-part-1-8ff164f78540. Accessed: 03.02.2019.

[40] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://www.bitcoinlightning.com/wp-content/uploads/2018/03/lightning-network-paper.pdf, 2016. Accessed: 03.03.2019.

[41] Raul. Transactions Speeds: How Do Cryptocurrencies Stack Up To Visa Or PayPal? https://howmuch.net/articles/crypto-transaction-speeds-compared. Accessed: 02.02.2019.

[42] Lukas Schor. Explained: Ethereum Plasma. https://medium.com/@argongroup/ethereum-plasma-explained-608720d3c60e. Accessed: 03.02.2019.

[43] Kai Sedgwick. No, Visa Doesn't Handle 24,000 TPS And Neither Does Your Pet Blockchain. `https://news.bitcoin.com/no-visa-doesnt-handle-24000-tps-and-neither-does-your-pet-blockchain/`. Accessed: 09.04.2019.

[44] Livio Sgier. Bazo – A Cryptocurrency From Scratch. `https://files.ifi.uzh.ch/CSG/staff/bocek/extern/theses/BA-Livio-Sgier.pdf`. Accessed: 02.02.2019.

[45] Stellabelle. Explain Delegated Proof Of Stake Like I'm 5. `https://hackernoon.com/explain-delegated-proof-of-stake-like-im-5-888b2a74897d`. Accessed: 03.02.2019.

[46] ZILLIQA Team. The ZILLIQA Technical Whitepaper. `https://docs.zilliqa.com/whitepaper.pdf`, August 2017. Accessed: 03.02.2019.

[47] Bozhi Wang, Shiping Chen, Lina Yao, Bin Liu, Xiwei Xu, and Liming Zhu. *A Simulation Approach For Studying Behavior And Quality Of Blockchain Networks*, pages 18–31. Springer, Cham, 06 2018.

[48] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling Blockchain Via Full Sharding. In *Proceedings Of The 2018 ACM SIGSAC Conference On Computer And Communications Security*, CCS '18, pages 931–948, New York, NY, USA, 2018. ACM.

Abbreviations

ABI	Actual block interval
ABS	Actual block size
BCS	Blockchain size
AggTx	Aggregation Transaction
AWS	Amazon Web Services
b04	Server of the University of Zurich on which one miner is running
calc.	Calcualted, theoretical possible value.
ConfigTX	BAZO specific transaction to change the blockchains overall configurations
FundsTX	BAZO specific transaction to send funds from one to another wallet
GB	Gigabyte
genB	Genesis block
GCP	Google Cloud Platform
IoT	Internet of Things
LFS	Large File System
lvB	Last validated and secure block
PoS	Proof-of-stake consensus technique
RAM	Random Access Memory
SIGBUS	Bus error [10]
StakeTX	BAZO specific transaction to change the staking behaviour of an account
TPS	Transactions validated in the network per second
$\text{TPS}_{\text{calc.}}$	Calculated, maximal possible TPS
TPS_{max}	Highest TPS during a test run
TPS_{min}	Lowest TPS during a test run
TPS_{sent}	Transactions sent to the network per second
Tx	Transaction
TxCnt	Transaction counter, counter which prevents double spending
uint64	unsigned 64-bit integer
uint8	unsigned 8-bit integer
VM	Virtual Machine
w/	with
w/o	without

Glossary

Altcoins Altcoins are alternative cryptocurrencies [14].

BoltDB GO specific key/value database [33].

Bootstrap Miner The bootstrap miner is the entry point to the network for a new joining miner. This Bootstrap miner does provide a list of other miners as well as the first few blocks. After the first couple of blocks, a connection to more miners should be established and the remaining blocks get also requested from them. This ensures a faster startup.

Bucket BoltDB specific collection of key/value pairs within the database [33].

Channel Channels are pipes which connect concurrent and synchronously running Goroutines. [36].

ConflictingBlockHashWithoutTx1 HashWithoutTransactions from the first conflicting block.

ConflictingBlockHashWithoutTx2 HashWithoutTransactions from the second conflicting block.

Defined blocksize The defined block size is the size which is set with a *ConfigTx* or in the code directly. This is the maximum size a block can have with all transactions and data included.

Fixed Blocksize In BAZO the fixed block size is 658 byte and consists of the fixed sizes form the block struct (393 byte), the common proof length (256 byte) and the maximum length of the bloomfilter plus one.

Fork A fork is a split in the chain. There are mainly three types of forks, further described in section 5.1

Genesis Block First block of a blockchain.

Goroutine A Goroutine is a lightweight thread which allows synchronously computation [37].

Git LFS Git Large File Storage (LFS) is used to store files larger than GitHub allows [4].

HashWithoutTx Hash Without Transactions.

Jq Jq is a command line JSON processor [5].

Lubuntu Lubuntu is a lightweight Linux operating system [6].

Mempool List, in which all open transactions are stored.

PrevHashWithoutTx Previous Hash Without Transactions

Rollback When an accidental fork occurs, the process of resolving these forks consists of a rollback phase and a validation phase. During the rollback phase, wrong validated blocks are revoked and in the validation phase, the new received chain branch gets validated.

Slice A slice is a GO specific data type which does describe a piece of an array [2].

Shardchain A part of a blockchain which is only validated in one shard.

Target Constant, which determines the speed of validating new blocks in BAZO [19].

TxHashSize In BAZO the transaction hashes have a length of 32 bytes.

TMUX TMUX is a terminal multiplexer for Linux [38].

Wallet A wallet is basically a *.txt*-file which contains a public and a private key for signing and signature verification of a transaction. The first two lines are the public key, the last line the private key [44]. Furthermore, the public key is also used as the address of an account in the state. In BAZO, coins are sent between wallets.

List of Figures

List of Tables

Appendix A

Installation And Usage Guidelines

A.1 Virtual Machines

This section does describe how the setup of virtual machines, miners and clients can be done on a new created Ubuntu VM. Since BAZO is a frequently changing research blockchain, it does not make much sense to create a fixed unchangeable virtual machine or docker container. The version of BAZO used for this thesis is not truly permissionless and therefore, a ready to use VM, which can be deployed to any cloud provider wanted, is not possible since the accounts are not correct. For the setup process, a bash shell script was created, which allows fetching the desired branch from GitHub. This is way more flexible.

There is a virtual machine, called *BAZO-VM* on the *CD*, which is ready to use with Oracle's Virtualbox. This Virtual machine has four miners and clients. It can be used to test the transaction aggregation locally. Further information about this VM can be found in section A.2.4.

Prepare New Virtual Machines

1. **Create wanted instance on desired cloud provider:** Depending on the desired cloud provider, different machines in terms of computation power, storage, operating system and various other selection possibilities are available. One or multiple virtual machines should be created according to the tutorials provided by eh cloud provider. Connecting to this instances via a *ssh connection* is recommended. The desired *TCP*-ports for BAZO (at the time of writing ports *8000*, *8001*, *8010* and *8020* are used) need to opened automatically.

2. **Prepare VM for BAZO:** Once connected to the VM, the *GO_setup_script.sh* from the GitHub repository *BAZO-Scripts-Fabio* (`https://github.com/febe19/BAZO-Scripts-Fabio`) can be used.
   This script installs *GO*, *TMUX*, *jq*, *git lfs* and an alias for the public IP-address

(*pubIP*) on the virtual machine. Furthermore it does an update of the operating Linux system, set the GOPATH and enables a colorful command prompt.

Steps:

(a) clone the *BAZO-Scripts-Fabio* repository with:
```
$ git clone https://github.com/febe19/BAZO-Scripts-Fabio.git
```
(b) change directory into *BAZO-Scripts-Fabio*

(c) run the *GO_setup_script.sh* with *source*, because otherwise, the coloring of the prompt does not work.
```
$ source GO_setup_script.sh
```
(d) if needed, insert [*Y*] during the execution of the script.

After steps *2.(a)* to *2.(d)*, the newly created machine is BAZO ready.

A.2 BAZO

The *README.md* from the miner and client application does describe the setup process perfectly.

The most important part here is that the correct *Wallets* and *Commitments* are used on every miner and client in the network, as otherwise, BAZO will never run correctly.

There are two types of a miner, the bootstrap miner and all other normal miners. Normally the bootstrap miner is driven by an account with root privileges.

The implementation and versions used for test runs are in these GitHub repositories:
Miner: `https://github.com/febe19/bazo-miner`
Client: `https://github.com/febe19/bazo-client`

In the client repository, the *performance_test_branch* should be taken.
On miner side, there are two branches, depending on what version is wanted. For a BAZO miner with transaction aggregation enabled, the branch (*TransactionsAggregationWorking* is the correct one. To test BAZO without aggregation but with all fixes, the branch *NoTransactionAggregation* should be used. In a network, all miners should run the same version, because otherwise, a hard fork may occur.

A.2.1 Setup Bootstrap Miner And Client

Setting up the bootstrap miner is a bit different from the others since no accounts are available at this point. Often the wallet with root access runs the bootstrap miner.

Steps:

1. git clone BAZO miner and client from desired fork with, *e.g.*:
```
$ git clone https://github.com/febe19/bazo-miner.git
```

2. change directory into *bazo-miner* and *bazo-client* and build projects with:
   ```
   $ go build
   ```

3. start miner in the *bazo-miner* folder with the following command. Thereby exchange the *publicIP* with the external IP-address of the machine:
   ```
   $ ./bazo-miner start --database StoreRoot.db --address publicIP:8000
   --bootstrap publicIP:8000 --wallet WalletRoot.txt --commitment
   CommitmentRoot.txt --multisig WalletRoot.txt --rootwallet
   WalletRoot.txt --rootcommitment CommitmentRoot.txt
   ```

4. in the client folder: replace the IP-addresses and ports in the *configuratons.json* file with the used IP-address and port. For the client running at the bootstrap miner, all IP-addresses are the public IP-address of the bootstrap miner.

5. copy *WalletRoot.txt* & *CommitmentRoot.txt* to the client folder.
   In the client folder, execute steps *5.(a)* to *5.(c)* for every new account. The name in *WalletName.txt* and *CommitmentName.txt* should be unique. Furthermore, the *txcounter* needs to be increased by one for every new account.
   Steps:

   (a) create new account with:
   ```
   $ ./bazo-client account create --rootwallet WalletRoot.txt
   --wallet WalletName.txt
   ```
   (b) add funds to new created wallet with: (*txcounter* needs to be increased)
   ```
   $ ./bazo-client funds --from WalletRoot.txt --to WalletName.txt
   --txcount 0 --amount 2000 --multisig WalletRoot.txt
   ```
   (c) enable staking for new account with:
   ```
   $ ./bazo-client staking enable --wallet WalletName.txt
   --commitment CommitmentName.txt
   ```

In the *BAZO-Scripts-Fabio* repository, a script called *AddAccountsScript.sh* does automate steps *4.(a)* to *4.(c)* whit slight different names, chosen for the network tests. The naming convention is described in section A.2.4.

A.2.2 Setup Normal Miner And Client

In the *BAZO-Scripts-Fabio* repository, a script named *MinerClientWithAccountCreated.sh* does help setting up a normal miner and client. This client is not conducted by a wallet with root access.

If the desired branch is on a special git fork, it needs to be added manually into the script code with the commands *git remote add forkName* `URL` followed by a *git pull forkName*. The *forkName* is user-defined and the `URL` is the URL of the desired repository. There are already two forks added. An additional one can be added analog. When this is added, the automated building will work also for additional forks.

If the *Wallet* and *Commitment* files are stored in the folder as mentioned in Tips & Tricks in section A.2.4, and these two folders are pulled with the *BAZO-Scripts-Fabio* repository, this script will copy the folders to the miner and client directory.

Steps:

1. execute *MinerClientWithAccountCreated.sh* with:
   ```
   $ ./MinerClientWithAccountCreated.sh
   ```
2. insert cloud provider and location and confirm both with [*ENTER*]
3. insert the git branch name, which should be used to build the miner application. Confirm first with [*ENTER*], and if prompted, insert [*Y*] to affirm.
4. same procedure for the client application: Insert the git branch name, which should be used to build the client application. Confirm first with [*ENTER*], and if prompted, insert [*Y*] to affirm.
5. if prompted, inset public IP-address of the bootstrap miner. This script will replace all IP-addresses and ports in the *configurations.json* correctly.
6. if the naming convention, presented in Tips & Tricks (section A.2.4, is satisfied, and there is a file with the proposed name, insert [*Y*] when asked. Otherwise copy the in step 5 of section A.2.1 created *Wallet* & *Commitment* files to the *bazo-miner* and *bazo-client* folder by hand.

At the end of the script and if it is used correctly, the start command of the miner is printed to the command line. When this command is executed in the *bazo-miner* folder, it should run.

A.2.3 Usage

The usage is exactly the same as in the *README.md* of a client. *E.g.* sending funds from the wallet *GCPSaoPaulo* to *GCPFrankfurt* is done with:

```
$ ./bazo-client funds --from Wallets/WalletGCPSaoPaulo.txt --to
Wallets/WalletGCPFrankfurt.txt --txcount 0 --amount 1 --multisig
Wallets/WalletB04Root.txt
```

A.2.4 Tips And Tricks

Here some tips & tricks for using BAZO in a distributed network as well as locally.

- Since every miner and client in the network, needs access to all *Wallet* and *Commitment* files, a logical naming scheme should be applied. A good one also used is: *Wallet + CloudProviderAbbreviation + Location* for wallets and the same for commitment files but starting with *Commitment*

- Exchanging the *Wallets* and *Commitments* to multiple machines works well with the help of GitHub. Therefore create all *Wallet-* and *Commitment-* files in folders called *Wallets* and *Commitments*. These folders are then copied to the *BAZO-Scripts-Fabio* repository (or any other) and pushed there for further usage.

- If a file called in the start command does not exist, BAZO creates it without letting the user know. This can cost a lot of time because small typos may be missed and a miner will not work as it should.

- For a local usage the *README.md* in the miner and client repository explain everything perfectly.

- When testing BAZO in a global network, it is recommended to start TMUX sessions for miner and client and execute the applications in there. This has multiple advantages, as it is possible to detach the sessions and move them to the background, which allows disconnecting from the VM. Furthermore, the process does not stop, if the connection is lost accidentally.

- BAZO only checks if a miner is bootstrapping according to the port. This results in a case, where two miners with different IP-addresses but the same port cannot connect to each other during startup. Therefore, it is recommended that the bootstrap miner uses a port not used from any other normal miner. The normal miners can have a similar port number but then with a different IP-address.

- Ensure that the IP-addresses and ports are correct. While starting a normal miner, the `--address` is the public IP-address of the machine the miner runs on, and the `--bootstrap` is the public IP-address of the bootstrap miner. In the *configuration.json* file of a client, the IP-address of *this_client* is the public IP-address and port of the client application. The *bootstrap_server* is the miner application to which the transaction should be sent. This does not have to be the bootstrap miner as setup in section A.2.1.

- When running multiple miners on one machine, either start the miners in different folders or ensure that the log files have distinct names. This can be enabled / disabled with commenting / uncommenting certain lines in the function `func InitLogger() *log.Logger {...}` (in file: *bazo-miner/storage/utils.go*) from the two in section A.2 mentioned branches.

- The *BAZO-VM* can be imported into Oracle's VirtualBox. This VM includes the latest version of BAZO with transaction aggregation and the emptying of blocks enabled. After importing, there is a document called *Tutorial.txt*, which describes how to use the installed miners and clients. The sudo password is *BAZO-VM*.

- It is recommended to start using BAZO on a local setup. Once BAZO runs locally, creating a global network should be easier.

Appendix B

Contents of the CD

- GitHub repository for the miner application,
  (*bazo-miner*)
- GitHub repository for the client application,
  (*bazo-client*)
- GitHub repository for the BAZO scripts,
  (*BAZO-Scripts-Fabio*)
- LaTeX source code,
  (*Evaluation_and_Improving_Scalability_of_the_BAZO_Blockchain.zip*)
- Final thesis,
  (*Evaluation_and_Improving_Scalability_of_the_BAZO_Blockchain.pdf*)
- Excel sheets for calculation,
  (*BCS_Test.xlsx*, *TPS_BlockInterval_Test.xlsx* & *TPS_BlockSize_Test.xlsx*)
- Midterm presentation & Aduno workshop presentation,
  (*Midterm_Presentation.pdf*, *Aduno_Workshop.pdf*)
- Virtual Lubuntu machine for local usage in VirtualBox,
  (*BAZO-VM.ova*)
- Abstract & Zusammenfassung,
  (*Abstract.txt*, *Zusfsg.txt*)
- Archive of used Log-files,
  (*Performance_Archive.zip*)