University of Zurich[UZH]

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

Design and Prototypical Implementation of a Mobile Light Client for the Bazo Blockchain

Marc-Alain Chételat
Zurich, Switzerland
Student ID: 10-915-718

Supervisor: Prof. Dr. Thomas Bocek, Bruno Rodrigues
Date of Submission: March 8, 2018

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

ifi

Master 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

Die Blockchain ist ein dezentrales Buchführungssystem, das keine vertrauenswürdige, zentrale Instanz erfordert. Transparente und unveränderbare Transaktionen können dabei automatisch prozessiert werden. In dieser Arbeit wurde ein *light client* für das *Bazo* Zahlungsverkehrssystem [4] entworfen und anschliessend eine Prototyp-Version implementiert. Der *light client* erlaubt es Benutzern - trotz beschränkter Speicher- und Netzwerkkapazitäten - Daten der Blockchain in einer mobilen Umgebung zu nutzen und am Netzwerk teilzuhaben. Ein Mechanismus, genannt *multi-signature*, erlaubt es Transaktionen innert Sekunden zu verifizieren. Dies macht das System vielseitig einsetzbar, beispielsweise in Läden oder Restaurants.

A blockchain is a trustless public ledger and allows the automatic process of transparent and immutable transactions. This thesis contains the design and prototypical implementation of a *light client* for the *Bazo* payment system [4]. The light client allows users to participate in the *Bazo* network in a mobile environment, where memory resources and network bandwidth are limited. Further, the system allows to verify transactions within seconds, using a *mutli-signature* mechanism. This makes the system usable for real-world on-the-go payment use cases, *e.g.* in shops or restaurants.

Acknowledgments

I want to express my sincerest gratitude to my supervisor Prof. Dr. Thomas Bocek for his competent and enthusiastic support during this thesis. With his immense knowledge and passion in the field of blockchains, he was able to help me during all times of research, implementing and writing this thesis.

Further I would like to thank Bruno Rodrigues for sharing his academical expertise with me and giving me positive feedback on my report.

Besides my supervisors, I would also like to thank Prof. Dr. Burkhard Stiller and the members of the Communication Systems Research Group for giving me the opportunity to work on such an interesting topic.

Last but not least I would like to thank my family and friends for their constant support and encouragement during my whole studies.

Contents

Chapter 1

Introduction

In 2008, a document called "Bitcoin: A Peer-to-Peer Electronic Cash System" was released under the name Satoshi Nakamoto. One year later, an open-source application was made public [15]. An alternative currency to fiat money, called *Bitcoin*, was issued and backed not by a central authority, but by automated consensus among networked users using the open-source application. In a precise and technical definition, Bitcoin is digital cash that is transacted via the Internet in a decentralized, trustless system using a public ledger called *the blockchain* [21]. Since 2009, digital currencies and the blockchain technology itself are on the rise. In February 2018, 1519 different cryptocurrencies were listed with a total market capitalization of over 430 billion US dollars [9].

Since the financial industry is aware of this emerging technology, they are pushing to get familiar with the blockchain technology and they are about to develop their own proof of concepts. So does a Zurich-based financial service provider which is active in the field of online payments. It developed a bonus program that incentivizes customers to use their credit cards by issuing virtual points for every conducted purchase. In a next step, the virtual points can be exchanged for *Bazo* coins by the finanical service provider's customers. *Bazo* is a blockchain-based cryptocurrency developed by the Communication Systems Group of the University of Zurich [4]. The goal of this thesis is to design and implement a light client solution for *Bazo* to enable customers manage accounts, send and receive *Bazo* coins in a mobile environment. This thesis is a highly explorative collaboration between the Communication Systems Group and the former described company.

1.1 Motivation

The blockchain technology provides *Bazo*'s public ledger, an ordered and timestamped record of transactions stored in blocks giving the global state. All blocks chained in the relevant order result in the blockchain. Since the network is organized in a decentralized manner, each client of the *Bazo* network independently stores the blockchain only validated by that client. Thus, to participate in this trustless network, a client must download and validate initially all the blocks contained in the valid chain. Since the

client holds the entire chain in memory, it is able to evaluate the system's state. For example, for each available account, it can calculate the corresponding balance in *Bazo* coins. A blockchain increases its size over time by nature. For example in January 2018, the Ethereum blockchain reached approximately 300 GB in size [7]. Hence, the device must meet minimum requirements regarding memory and network capacities in order to manage the blockchain's data. Requirements state-of-the-art mobile phones cannot meet. But limiting this functionality only to clients using maschines with strong computational and storage power, *e.g.* stationary computers, is nowadays not reasonable. Therefore the concept of a light client must be ellaborated, designed and implemented for *Bazo*.

The goals of this thesis are to do research on how to reduce data in a blockchain and integrate it into a light client application while keeping the data's consistency. The system must be evaluated and tested in an environment with real customers.

1.2 Description of Work

This thesis covers the development of a light client for the *Bazo* cryptocurrency. The outline will be an "adapter" for third party applications to the *Bazo* network validating only particular data affecting the system's state the client is interested in. For example mobile wallets want to know one, two maybe three account addresses' states and are therefore only interested in a limited amount of transactions (and not all of them). Thus, only the transactions relevant to the account's state must be considered for validation. This thesis ellaborates techniques to validate relevant transactions and to feed the network with transactions in a fast and reliable way in a mobile environment.

1.3 Thesis Outline

The focus of this work is on running a prototype that will be tested and evaluated in the financial service provider's testing environment with real customers. In Chapter 2 the blockchain technology as well as full and light clients will be introduced. The following Section 2.1 will describe the *Bazo* protocol's key properties on which the light client is built-on. Further techniques used in the following Chapters will be elaborated. In Chapter 3 the financial service provider's requirements and its implications for the prototype are documented. Chapter 4, 5 and 6 outline the design, implementation and evaluation respectively. Whereas Chapter 7 finally summarizes and concludes this thesis. Finally a lookout about future work will be given.

Chapter 2

Background & Related Work

This Chapter introduces briefly the blockchain technology with a focus on full and light clients respectively. The second Section describes the *Bazo* blockchain in order to understand on what basis the light client is built-on.

2.1 Blockchain Technology

The blockchain technology is a decentralized system of chained transactional records. A subset of network participants (also known as *miners*) enrich the chain by solving difficult computational problems. Miners compete anonymously on the network to solve the mathematical problem in the most efficient way. The successor adds the next block to the chain and obtains a block reward (*i.e.* newly minted coins) as an incentive. The transactions in the minted block are considered to be valid. All transactions ever validated represent the *public ledger*, an immutable collection of transactions. If system-wide mining power increases, so does the difficulty of the mathemmatical problems required to mine new blocks (assuming proof-of-work mechanism) [17].

2.1.1 Transactions

Blockchain technology ensures to get rid of the *double-spend problem* [8] with the help of cryptography. Each user is assigned a public and a private key. The public key is a cryptographically generated address and can be shared among other users. The private key is kept secret by the user (like a password). Every coin is associated with an address. A transaction is simply a trade from one address to another and it is initiated if a user (future owner) sends its address to the coin holder. The transaction is hashed and digitally signed by the sender's private key. Since the transaction contains the receiver's public key, only the user with the corresponding private key matching the public key is able to spend the coins (Figure 2.1). An important feature is that public keys are never tied to real-world identities. However, transactions are enabled without revealing the involved parties' identities and are still traceable. This is a major difference to transactions between common financial institutions [17].

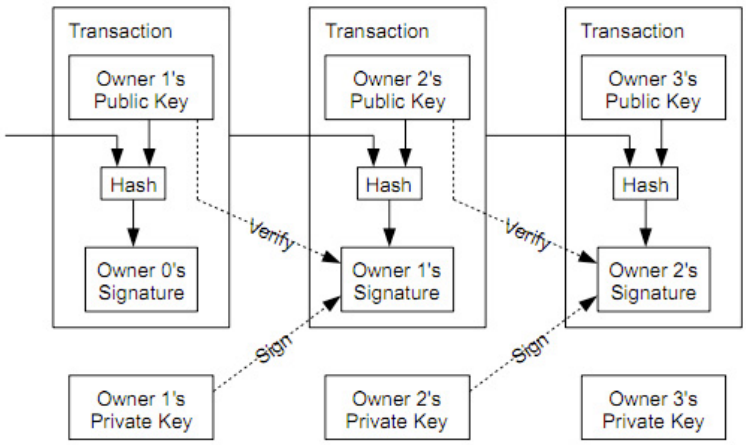

Figure 2.1: Blockchain-based transactions [15]

2.1.2 Chain of Blocks

The blockchain relies extensively on hashes and hash functions. A hash function is a deterministic mathematical algorithm transforming an input into an output. It is characterized by its extreme difficulty to revert. Thus, it is (almost) impossible to revert the output in its original input. This is called the collision resistance [17].

In the blockchain, hash functions are used to hash block's data. One or more transactions are assigned to a block. Copies of transactions are hashed, and the hashes are then paired, hashed, paired again and hashed again until a single hash results, the so-called *merkle root*. This merkle root will be saved within the block together with the previous block's hash value and other arbitrary block data (*e.g.* timestamp, ...) (Figure 2.2). A block saving the predecessor's hash value chains the blocks together. This ensures a transaction cannot be modified without modifying the block that records it and all proceeding blocks [6].

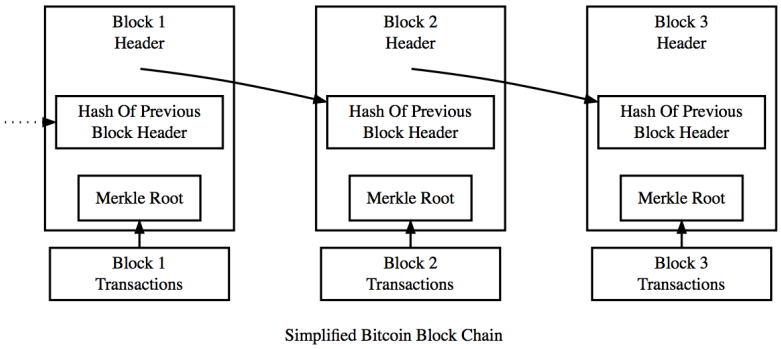

Figure 2.2: Simplified blockchain [20]

2.1.3 Full Client

Since the blockchain stands as a trustless proof mechanism of all transactions on the network, users can trust the system of the public ledger stored worldwide on different decentralized *full clients* (also denoted as *nodes*) maintained by "miner-accountants". Each full client (*i.e.* every computer connected to the network using a client that performs the task of validating transactions) has a local copy of the blockchain, which is downloaded automatically when the full client joins the network [21].

2.1.4 Light Client

In 2016, digital services have been shifted twelve percent to mobile environments within a year and the trend continues [13]. Therefore applications for blockchains must be working in mobile environments. Unfortunately, the participation of mobile clients was not originally intended by the creators of the blockchain architecture. As mentioned earlier, the Ethereum blockchain's size would nowadays exceed a regular mobile phone's memory capacity, referring to only one problem. In order to make participation possible for technically less powerful clients, mechanisms must be developed to meet the devices technical constraints. Therefore the light client protocol is introduced. Its purpose is to allow users to obtain reliable, secure information on their accounts by verifying only all relevant transactions. A light client cannot mine new blocks.

2.2 Bazo

Bazo is a blockchain-based cryptocurrency developed at the Communication Systems Group at the University of Zurich together with the mentioned financial service provider. The goal is to create a decentralized, simplified payment system for the financial service provider's bonus system. *Bazo* uses an account-based data model where transactions lead to account state changes. All accounts of the *Bazo* blockchain united make up the public ledger. All of this Section's content refers to "Bazo - A Cryptocurrency from Scratch" [18].

2.2.1 Accounts

An **account object** has the following structure:

Address The address of the account is the public key of an elliptic curve key pair.

Balance The balance represents the amount of *Bazo* coins residing in the account.

Transaction Counter The transaction counter is used to prevent replay attacks. This counter is incremented in the sender's account for every transaction that involves the transfer of coins.

2.2.2 Transactions

Transactions are the basis for every state change. *Bazo* takes three different transactions to manipulate the state. Transactions for account creation, transferring funds and changing system parameters. All transaction types contain a fee and a signature. The **fee** is payed by the creator of the transaction and incentivizes the miners to include the transaction in the next block. The **signature** is for authentication and validation purposes.

Users cannot join the *Bazo* network voluntarily, since it represents a private/invite-only blockchain. Before a new user can transfer coins, a root user must create a new account. Account creation transactions are referred to as *AccTx* for the rest of the thesis.

An **account creation** transaction has the following structure (Table 2.1):

Transaction	Property	Description
AccTx	Header	Reserved for later use
	Issuer	The root's public key (hash)
	Address	The newly created account's public key
	Fee	The fee payed for the transaction
	Signature	The transaction's signature

Table 2.1: Overview of the *AccTx*'s structure

In order to transfer coins from one account to another, funds transaction (from now on referred as *FundsTx*) are introduced. *FundsTx* are valid, if the transaction was signed by the sender's private key, the sender's balance is equal or greater than the amount spent and the sender's transaction counter matches the state.

A **fund transferring** transaction has the following structure (Table 2.2):

Transaction	Property	Description
FundsTx	Header	Reserved for later use
	Amount	The amount to be transfered
	Transaction Counter	The sender's transaction counter
	From	The sender's public key (hash)
	To	The receivers's public key (hash)
	Fee	The fee payed for the transaction
	Signature	The transaction's signature

Table 2.2: Overview of the *FundsTx*'s structure

Using the system parameters transaction, the *Bazo* protocol's system parameters can be changed at runtime and hard forks can be avoided. A system parameters transaction is only valid, if it was signed by a root account. System parameters transaction are denoted as *ConfigTx*.

A **system parameters** transaction has the following structure (Table 2.3):

Transaction	Property	Description
ConfigTx	Header	Reserved for later use
	Id	The parameter to be changed
	Payload	New value for the chosen parameter
	Transaction Counter	The sender's transaction counter
	Fee	The fee payed for the transaction
	Signature	The transaction's signature

Table 2.3: Overview of the *ConfigTx*'s structure

Table 2.4 describes all system parameters, which can be changed using a *ConfigTx*.

Id	Property	Description
1	Block size	This parameter stands for the maximum block size (in bytes). Received blocks that are larger than this parameter get rejected by the miners.
2	Difficulty interval	For stability reasons the difficulty of the proof-of-work should be relative to the hashrate of the *Bazo* system. This parameter indicates the amount of blocks that are to be validated before a new target value is calculated.
3	Fee minimum	To incentivize miners for their mining work, the minimum fee that a transaction needs to pay can be set.
4	Block interval	The block interval is a time parameter that describes how much time (in seconds) shall pass between two blocks.
5	Block reward	To further boost miner incentives, an additional block mining reward can be set with this parameter.

Table 2.4: System parameters which can be changed using a *ConfigTx*.

2.2.3 Blocks

Transactions themselves do not change the system's state, but they are consolidated to blocks. If a block is validated successfully, it is added to the chain and the system's state is updated. The main reason for having transactions consolidated within blocks are system stability and consensus.

A **block** has the following structure (Table 2.5):

Property	**Description**
Hash	The block hash acts as a unique identifier of blocks within the blockchain.
Previous Hash	This value is equal to the identifier of the previous block in the blockchain.
Nonce	The nonce is set to a value such that the resulting hash fulfills the proof-of-work requirements.
Timestamp	Refers to the block creation time (seconds elapsed since January 1, 1970 UTC).
Merkle Root	The value of the merkle tree's root node.
Beneficiary	The address hash of the account that receives fee payments and the block reward.
Nr. AccTx/FundsTx/ ConfigTx	Corresponds to the number of transactions of each type that are included in the block.
Hash Data AccTx/ FundsTx/ConfigTx	The hashes of all transactions included in this block in sequential order.

Table 2.5: Overview of a block's structure

A block's size depends on how many transaction hashes are saved within the block's body. The header's size is 149 bytes (Figure 2.3).

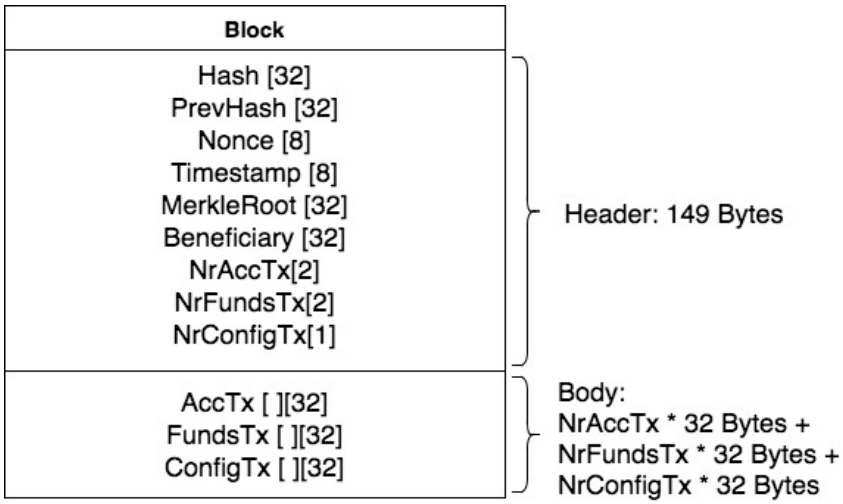

Figure 2.3: Block divided into header and body [18]

In contrary to most blockchains, only the transaction's hash values are saved in blocks. The goal is to keep the block size as small as possible. A block is considered to be valid and added to the chain if all of the following requirements are met:

- All included transactions must be structurally valid and their sequential execution must lead to legal state changes.
- There must be no transaction duplicates within the block.
- There must be no transaction that was already validated in a previous block.
- The block must belong to the longest chain at the time of validation.
- The timestamp of the block must be within a predefined timerange of the validation time.
- The block size must not be larger than the current block size parameter.
- The beneficiary account must exist in the state.
- The merkle root must be correctly calculated.
- The proof-of-work must satisfy the consensus properties.

2.3 Bloom Filter

A bloom filter is a space-efficient, probabilistic data structure and tests if an element is a member of a set or not. False positive matches are possible, false negatives are not [1]. For example, a query returns "element is possibly in the set" or "element is for sure not in the set". Instead of a bloom filter, hash maps could be used for efficient lookups, but bloom filters are additionally space-efficient. A bloom filter is a bit array of m bits, where m denotes the array's length. Another parameter is the number of hash functions k. Each hash function sets the bits in the array. When inserting an element s_i into the bloom filter, all the bits $h_1(s_i), h_2(s_i), ..., h_k(s_i)$ are set.

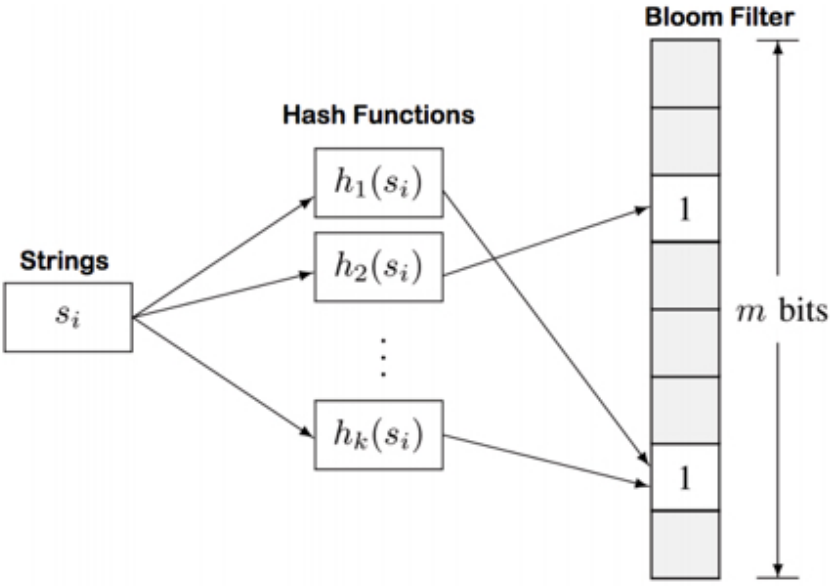

Figure 2.4: Bloom filter with $m = 8$ [20]

Example 1. Figure 2.4 indicates how an element s_i is hashed and the bits flipped in a bit array with $m = 8$ bits. The hash functions flip the second and the sixth bits whereas hash collisions are allowed (second bit). Querying the bloom filter for the existence of the element s_i will hash it again. If the hashes correspond to the flipped bits in the array, the result will be positive ("element is possibly in the set") (Figure 2.4).

Since hash collisions are accepted, false positives might occur. The false positive-rate depends on the array's length m and the number of hash functions k. This leads to the following trade-off: As the size of the bloom filter m increases, the occurrence of hash collisions for multiple items that are inserted into the bloom filter drops and the false positive-rate decreases [2]. In the extreme case no hash collisions at all occur and each element of the set is mapped to a unique bit. But at the same time the bloom filter increases in space.

The functions for choosing the bloom filter's size m and the quantity of hash functions k for achieving a certain false positive-rate p can be written as:

$$m = ceil(\frac{n * log(p)}{log(\frac{1}{2^{log(2)}})}) \tag{2.1}$$

and

$$k = round(log(2) * m/n) \tag{2.2}$$

with n representing the known number of elements in the set [19] (Example 2).

Example 2. Assumed $n = 100$ elements should be saved in the bloom filter with a false positive-rate of $p = 0.1$, m and k will be calculated as shown in Equation 2.3 and Equation 2.4.

$$m = ceil(\frac{100 * log(0.1)}{log(\frac{1}{2^{log(2)}})}) = 480 \tag{2.3}$$

according to Equation 2.1 and

$$k = round(log(2) * 480/100) = 3 \tag{2.4}$$

according to Equation 2.2. Saving 100 account addresses each having a size of 32 bytes (see Subsection 2.2.1) requires $100 * 32 = 32000$ bytes of space, compared to $\frac{480}{8} = 60$ bytes needed for the bloom filter (taking a false positive-rate of 0.1 into account).

Chapter 3

Requirements

The following use case was defined by the financial service provider: Customers must hold a specific credit card issued by the financial service provider and participate in a bonus program. The bonus program allows customers to collect bonus coins which then can be transferred to *Bazo* coins.

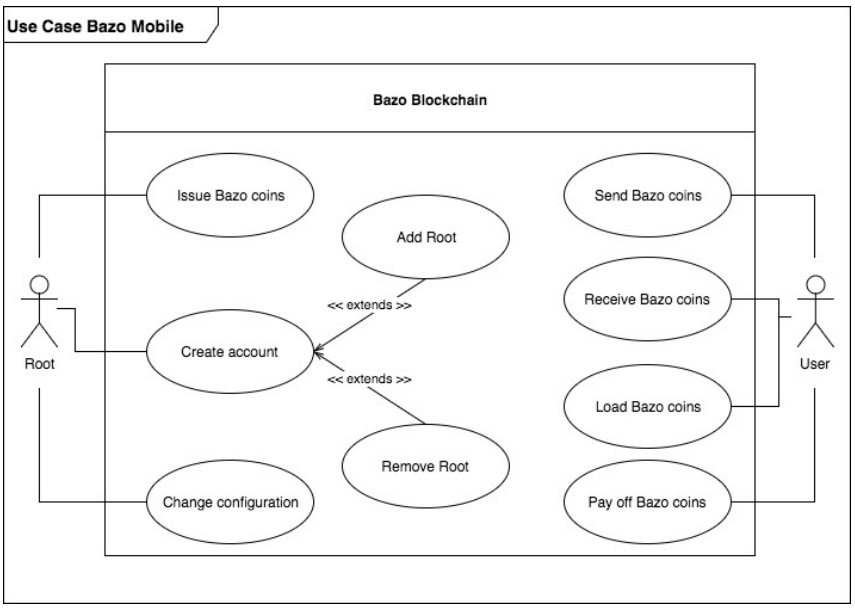

Figure 3.1: Use cases for *Bazo*

As Figure 3.1 indicates for *Bazo*, root accounts must be able to create new user accounts. Either they are regular or root users. By the time an account is created, the account's balance is zero. A root account has the permissions to initially fund another account without having coins substracted from its balance. Root accounts are also able to adjust system parameters.

Regular users initially convert bonus coins into *Bazo* coins. Therefore a functionality must enable users to load *Bazo* coins into their wallet, issued by a root. If not needed anymore, *Bazo* coins must be able to be converted back into fiat money and sent to a regular bank

account. The system's core functionality is sending and receiving *Bazo* coins to and from other users respectively. Since the pilot project aims to test the system in the financial provider's cafeteria, users must be able to pay with *Bazo* on-the-go and the receiver of funds must be able to verify the corresponding transaction within three seconds. Thus, the wallet with its functionalities to show account details, send and receive *Bazo* coins must be working on mobile devices. Taking the current version of *Bazo* [18] into consideration, the following problems described in Sections 3.1 and 3.2 can be identified:

3.1 Mobile Environment

Every client needs to download the whole blockchain data in order to participate. Since *Bazo* adds a new block to the chain constantly in a defined time interval, the blockchain grows indefinitely, even if no transactions were included in the added block [18]. Thus, the *Bazo* blockchain could reach several hundred gigabytes, such as the Ethereum blockchain already did [7]. At this point, a mobile client hits its technical constraints offering between 32 and 256 GB of disk memory and usually 2 GB random-access memory. Further national-wide network services limit downloading these amounts of data in a mobile environment, providing 20 to 30 Mbit/s in Switzerland [16].

3.2 Transaction Verification

The financial service provider requires a transaction to be verified within three seconds, but blockchains are not designated to verify transactions immediatly. A transaction is considered to be valid if the block it is saved in is successfully added to the chain and minted by the network. For example the Bitcoin blockchain adds a new block every seven to eight minutes in average, the Ethereum every 16 seconds [7]. Creating a block every three seconds in order to meet the requirement is not an option, since this leads to chain forks which must be resolved [18].

Thus the light client must fulfill the following requirements:

1. A mobile device must be able to calculate one or more accounts' states and display relevant account information such as the address and balance.
2. A mobile device must be able to send funds to the network in a reliable and secure way.
3. A mobile device must be able to verify the funds received within three seconds.

Chapter 4

Design

Even if the *Bazo* protocol optimizes memory management by design, it does not make sense to download the whole blockchain if the client is only interested in a subset of transactions. For instance if a user wants to know its balance, it does not need to know all transactions ever saved in the blockchain, but those which are relevant to the user's account. Obviously, account information could be obtained by requesting it from a full client. Since the blockchain is a trustless system, this solution is not an option. Every account creation was processed in an *AccTx* initially. *FundsTx* must follow after, processing sending and receiving coins. This chapter outlines the ligth client's design for *Bazo* according to its requirements described in Chapter 3. First a refactored block structure is introduced, altering the original *Bazo* block's [18] header and body. Afterwards a solution is presented to calculate an account's state, taking transaction verification and scalability into consideration. Finally a mechanism called *multi-signature* will be introduced to verify *FundsTx* within the required time limit (three seconds).

4.1 Block Header

The block structure introduced in Subsection 2.2.3 is refactored and extended with the following properties:

- A bloom filter and;
- the corresponding bloom filter's size.

Figure 4.1 illustrates the block's partition into a header and a body part. The block's hash, previous hash and *NrConfigTx* parameters are kept in the header. The newly introduced bloom filter and its size are added. The header contains now all information needed to evaluate if this block is relevant for an account's state calculation or not. The header has a minimum size of 67 bytes.

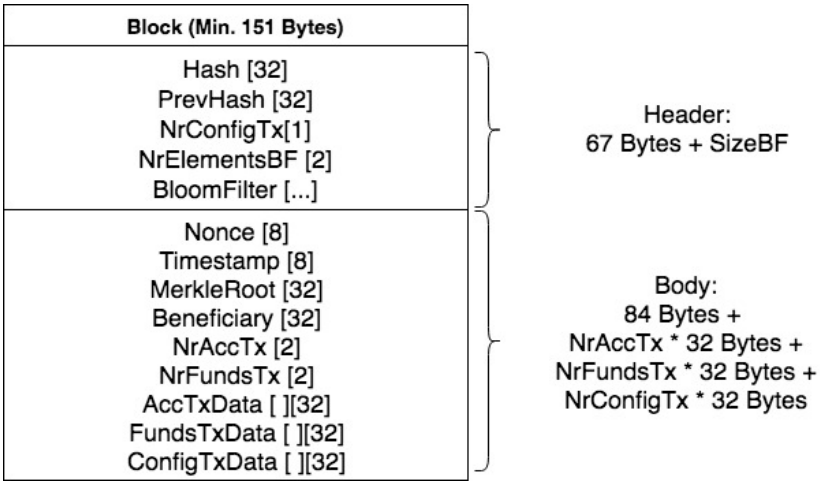

Figure 4.1: Refactored block structure

4.2 State Calculation

The light client needs to keep track of the following transactions for all relevant accounts a_m, where $m \in \mathbb{N}$:

- The one *AccTx* the account a_m and its address were created.
- All *FundsTx* the account a_m has been involved in (From/To).
- All *ConfigTx* changing system parameters.

Since the system parameter *block reward* is relevant for an account's state calculation (*i.e.* its balance), the light client needs to keep track of all *ConfigTx*. To calculate an account's state, the block containing the initial *AccTx* and all blocks containing relevant *FundsTx* as well as *ConfigTx* must be downloaded. The *AccTx* can be identified by the address property. All relevant *FundsTx* contain the account's address in the From/To fields (see Subsection 2.2.2). One approach of requesting the relevant blocks from the network is handing over the account's address and letting the network send the relevant blocks. By submitting the address to the network, privacy violations must be taken into consideration. While sending the account address from a device with an allocated IP-address the user's real identity could be revealed and the networks anonymity feature would be compromised. Therefore the better approach is to let the client search for relevant blocks and afterwards request them from the network. As introduced, the download of all blocks should be avoided. At this point, the light client only downloads all block headers - a selection of parameters and no transaction data (Figure 4.1). In order to separate the relevant from irrelevant blocks, all involved addresses - more specific their hashed values - are saved in a space-efficient, probabilistic data structure called bloom filter (see Section 2.3). As discussed in the beginning of this Section, an account is involved in the block and its address is saved in the bloom filter if at least one of the following conditions is met:

- The block includes the *AccTx* the account was created.

- The block includes a *FundsTx*, the address belongs either to the sender or to the receiver account.
- The block's beneficiary is the account.

Thus, a block is relevant for the light client if:

- The bloom filter's test of a particular address is positive.
- The *NrConfigTx* is greater than zero.

Hence, the light client requests all relevant blocks containing all transaction data from the network. The account's state can be calculated with all information available. As explained in Section 2.3, the trade-off for space-efficiency is the false positive-rate of bloom filter testing. For *Bazo*, a false positive-rate of 10% is given. Thus, in 10% of all cases, the light client requests an irrelevant block. This might be a disadvantage but keeps privacy alive. Considering the case if the bloom filter only contains one address, with a false positive-rate of zero, the network knows the requester's address. With a false positive-rate of 10%, the network cannot conclude the very same.

4.3 Block Verification

Bazo is a trustless system and the light client must not rely on received data sent by the network. The light client cannot assume the block's data to be valid. It needs to check the block's integrity and verify it by itself. Merkle trees provide efficient and secure verification of large amounts of data [14]. The process of data verification is simplified and scalable. Using merkle trees, the light client can easily check the block's integrity and the existence of a particular transaction in the block. Because all transactions in the block are part of the merkle root hash calculation, no transaction can be added or omitted after the merkle root has been calculated. Even the transaction order cannot be changed afterwards. A particular transaction's verification requires the merkle root, obtained from the block header, and a list of intermediate transaction hashes from a full client. The full client does not need to be trusted. The intermediate hashes cannot be faked or the recalculation of the merkle root will fail [6]. Example 3 illustrates how the light client verifies H_K.

Example 3. To ensure the existence of H_K, the light client must obtain H_L, H_{IJ}, H_{MNOP} and $H_{ABCDEFG}$ in sequential order (Figure 4.2). The nodes H_{KL}, H_{IJKL}, $H_{IJKLMNOP}$ and the merkle root $H_{ABCDEFGHIJKLMNOP}$ are recalculated by the light client based on the obtained intermediates. If the calculated merkle root matches the block header's merkle root, the transaction is in the block. The opposite case, however, is not true: If the recalculated merkle root does not match, it does not proof the inexistence of H_K.

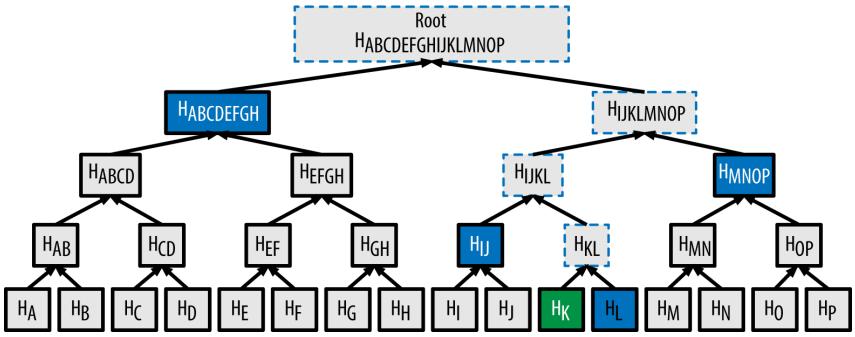

Figure 4.2: Merkle root verification of H_K [3]

4.4 Scalability

Scalability is a key feature for *Bazo*. The *Bazo* blockchain was designed for scalability and simplicity [18]. Thus, the light client must cope with scalability as well in order to avoid being a bottleneck for applications. The light client must work for multiple accounts with an arbitrary long blockchain. Forecasting the future key values such as chain length and transaction throughput are difficult to make. The Ethereum blockchain's properties are taken as benchmark values, since Ethereum is at that time one of the most efficient blockchains [7].

By extending the block header with a bloom filter, the header size becomes variable. Its minimum size is 67 bytes, if no transactions are available and the bloom filter is empty. If transactions are available, the block header size depends on how many accounts are involved into all transactions. The following calculations ensures, that the light client design is scalable:

In *Bazo*, a new block will be minted every minute in avergage [18]. In the early January 2018, Ethereum verifies 700 transactions per minute and the whole chain contains around 1 million blocks [7]. If *Bazo* verifies 700 transactions per minute at peak time, 1'400 addresses are stored in a block header at maximum (two for each transaction). Using Equation 2.1, to calculate the bloom filter's size, a block header results in almost 1 KB. Downloading initially 1 million block headers, each of 1 KB in size results in 1 GB total. With an average mobile data connection speed of 25 Mbps [16], it takes approximately five minutes to download all block headers.

4.5 Multi-signature

As required by the financial service provider's use case defined in Chapter 3, a user must be able to verify a transaction within three seconds. As mentioned earlier, minting blocks every three seconds is not an option. Therefore *multi-signature* is introduced. With multi-signature all *FundsTx* must be signed twice in order to be valid: By the sender as yet and by a third party additionally. The third party is a counterparty-server, designated only for multi-signing *FundsTx*. It keeps track of an account's open transactions and

decides if an account is solvent for incoming spendings or not. If the account is solvent, the multi-signature server signs the transaction and the network will be able to verify it. Therefore, the existing *Bazo FundsTx* must be extended by a second signature. The adapted protocol is illustrated in Figure 4.3.

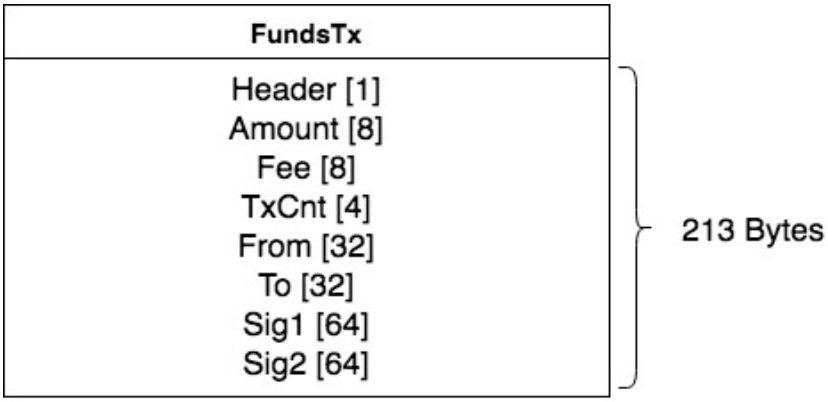

Figure 4.3: Extended *FundsTx*-protocol by second signature (Sig2)

First, the sender creates a *FundsTx* and signs it with its private key. After sending it to the light client, the *FundsTx* is sent to the multi-signature server with the sender's balance. The multi-signature server calculates whether the sender has enough funds considering all open transactions. Open transactions are not yet saved in a block and verified by the network. If there are enough funds, the multi-signature server signs the transaction a second time and sends it back to the light client. The light client then sends the transaction into the network and to the fund's receiver. The receiver is able to verify the second signature by itself and can assure, the sender has enough funds to spend the amount, although the transaction has not yet been verified by the network. In order to make sure the *FundsTx* reaches the network, the receiver can send the *FundsTx* a second time (Figure 4.4).

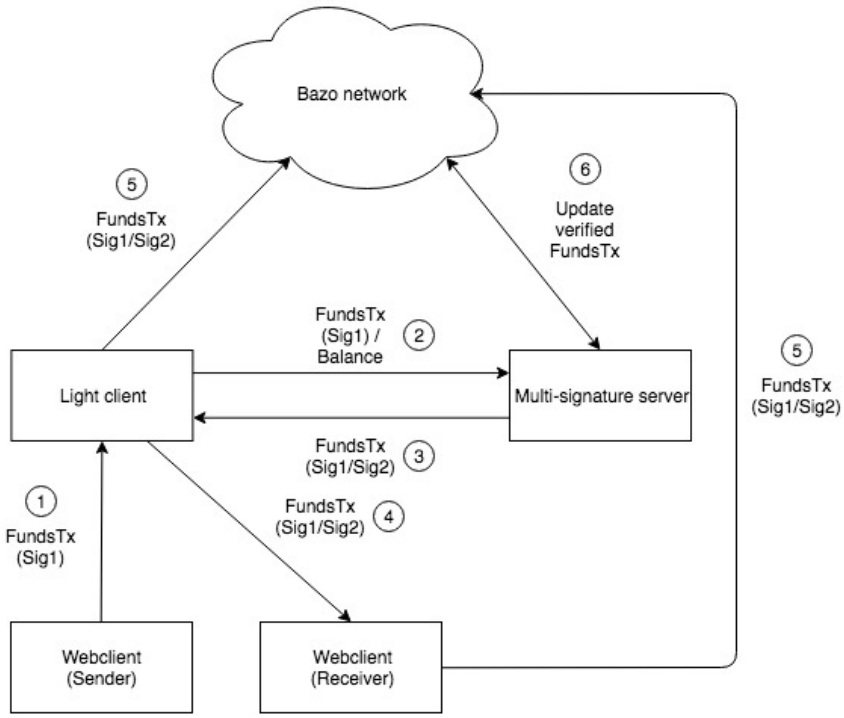

Figure 4.4: Integration of the multi-signature server

Chapter 5

Implementation

First, an overview about the light client's program structure is given. The Section following describes how the light client connects to the network in order to fetch all block headers. Then, the one and only new object the light client uses is described: The *Account* object. Afterwards an overview how the relevant blocks are elaborated is given, followed by Section 5.5 explaining how the light client can verify block and transaction data. Section 5.6 describes how the light client calculates an account's state. Since third-party applications - *e.g.* a web client - must be able to connect to the light client, Section 5.7 introduces a *REST* API. The last Section 4.5 illustrates how the multi-signature mechanism is implemented.

5.1 Program Structure

The light client is written - as the *Bazo* miner application (*bazo-miner*) - in *Go* which is a programming language supporting networking and concurrent programming [12]. The program is called *bazo-client* within the *Bazo* project [4]. Since in Go, import cycles are not allowed, the program must meet a predefined structure, illustrated in Figure 5.1. The program includes two packages; called *client* and *REST*. Further it includes packages from the *Bazo* miner application. These are mainly protocols, utility functions and the IP address for the connection to the *Bazo* network.

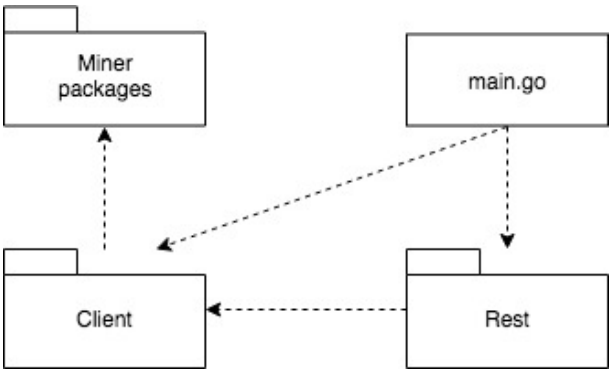

Figure 5.1: *Bazo* light client's program structure

The light client can be started by calling *main.go*. Installation and startup guidelines are described in Appendix A. Three different functionalities are provided, depending on the user's needs:

1. Start for sending an *AccTx*, *FundsTx* or *ConfigTx* to the network.
2. Start for requesting an account's state.
3. Start the light client's REST interface for handling incoming requests.

Sending transactions to the network and requesting an account's state (functionalities 1 and 2) over the *CLI* are mainly for admin purposes only. These functionalities can be accessed over the REST API too. Starting the latter is for serving third-party applications using the light client for accessing the network. The functionalities 2 and 3 do state calculation for a given account and therefore need to download all block headers from the current chain.

5.2 Downloading Block Headers

On startup - for functionalities 2 and 3 - the method *InitState()* is called. Figure 5.2 illustrates how all block headers are requested from the network. Since the light client receives block headers reverse chained [18], the last block header will be saved first in the light client's memory. In order to calculate an account's state, all transactions must be processed time-ordered, starting with the oldest. The reason is that before *FundsTx* can exist for a certain account, an *AccTx* must have created it. Thus, the block header's sequencial order must be inverted. All block header's are finally stored in the light client's RAM. For instance, they are not persisted. This implies redownloading all block headers again if the light client shuts down. After downloading all headers up-to-date, the light client starts a goroutine [12], a leightweighted thread of execution. It will download every ten seconds the headers of the new minted blocks recursively. The execution happens concurrently with the calling one.

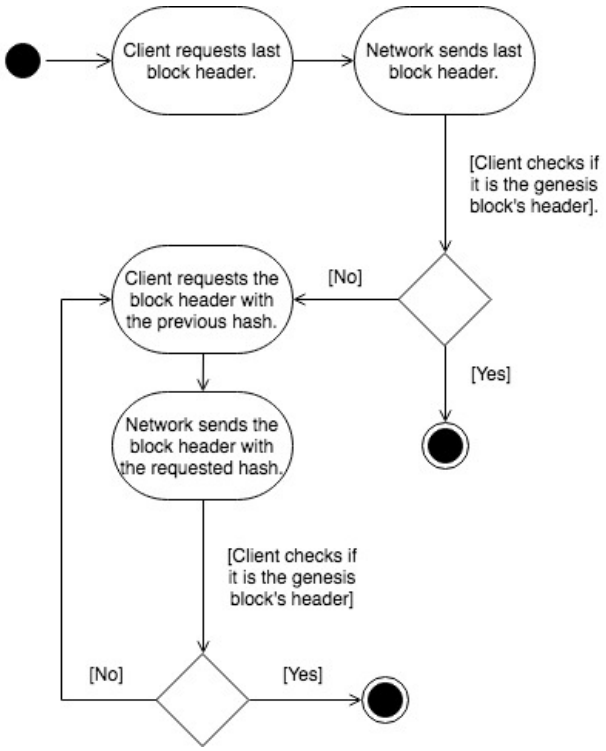

Figure 5.2: UML activity diagram representing how block headers are requested from the network

5.3 Account Object

The light client maintains an account object for each request calculating an account's state. Afterwards, the object is discarded again. The account object's structure is defined in *account.go* in the client package and implies the following fields:

- Address as *[64]byte*

- Address as *string*
- Balance as *uint64*
- TxCnt as *uint32*
- IsCreated as *bool*
- IsRoot as *bool*

The address field must be represented as a string additionally, in order to be readable in *JSON*, used as the REST's resulting format (Section 5.7). Further information about the balance, the account's transaction counter [18] and two boolean values representing if the account has been created and if it is a root account respectively are saved in the account object.

5.4 Elaborating Relevant Blocks

If an account's state is calculated, the light client first checks all block headers for their relevance. As in Section 4.2 described, a block is relevant if the bloom filter test for a certain address is positive or if the block contains at least one *ConfigTx*. The process is illustrated in Figure 5.3. The relevant blocks are requested from the network over a TCP-connection. The blocks are not cached and the next time the same account's state is requested, all relevant blocks have to be elaborated and requested from the network again.

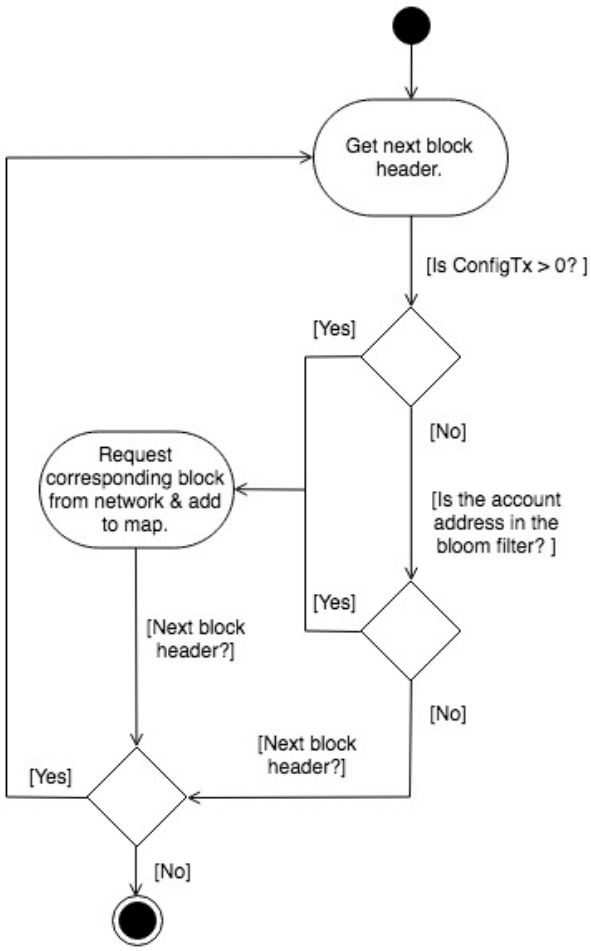

Figure 5.3: UML activity diagram representing how block headers are tested for relevance

5.5 Block Verification

The light client only verifies blocks which are elaborated as relevant according to Section 5.4. As introduced in Section 4.3, a block's merkle root is saved within the block and helps to verify the block's transactions and to assure the block's integrity. The merkle root is calculated out of all transaction's hash values, which represent the leafs in a binary tree

(Figure 4.2). The following subsections describe how building a merkle tree is implemented and how either the whole tree or only individual leafs can be verified.

5.5.1 Merkle Tree Build

In the miner applicaiton's code, the *newTree()* method in *merkletree.go* accepts all transaction's hash values as input arguments which are considered to be saved in the corresponding block and returns finally the root and leaf nodes of the merkle tree. Intermediate nodes are built in order to calculate the root's hash value and denote all nodes between the root and the leafs. If no transactions are saved in a block, the root's hash value is zero. A *Node* object has the following attributes:

- Parent as **Node*
- Left as **Node*
- Right as **Node*
- leaf as *bool*
- dup as *bool*
- Hash as *[32]byte*

An overview of the possible relations between root, leaf and intermediate nodes' attributes is given in Table 5.1:

Node	**Parent**	**Left**	**Right**	**leaf**	**dup**	**Hash**
Root	-	Interm.	Interm.	false	false	[32]byte
Interm.	Root/Interm.	Interm./Leaf	Interm./Leaf	false	false	[32]byte
Leaf	Interm.	-	-	true	true/false	[32]byte

Table 5.1: Possible relations for root, leaf and intermediate nodes

In a first step, all leaf nodes are built. A leaf's hash value equals the corresponding transaction's hash value. If the number of transactions is odd, the last leaf's copy is added in addition. Then all intermediate nodes are built recursively in *buildIntermediate()* (Algorithm 1).

Algorithm 1: Building the root and intermediary nodes

```
Data: Array of nodes to build intermediates from; nodes
Result: Array of intermediate nodes or root node; newNodes
l := len(nodes);
while l is not power of 2 do
    l = l - 1;
for i := 0; i < l; i = i + 2 do
    // Generate the new parent's hash from bottom up.
    parentHash = nodes[i].Hash + nodes[i+1].Hash;
    parent = Node{Left: nodes[i], Right: nodes[i+1], Hash: parentHash};
    // Add the new parent to the result.
    newNodes = append(newNodes, parent);
    nodes[i].Parent = parent;
    nodes[i+1].Parent = parent;
    // Abort recursive method if the root node is generated.
    if l == 2 then
        return newNodes;
// Add the remaining nodes not considered to the next level.
if l < len(nodes) then
    for i := l; i < len(nodes); i++ do
        newNodes = append(newNodes, nodes[i]);
// Recursive call
buildIntermediate(newNodes);
```

5.5.2 Merkle Tree Verification

For verifying a block's merkle root, the whole tree must be available or rebuilt. The method *verifyNode()* checks the tree recursively and recalculates the hash for the requested node. Later, the hash values can be checked for equality. Algorithm 2 illustrates the function:

Algorithm 2: Verifying a node's hash value, if merkle tree is available

```
Data: node
Result: hash
// The bottom of the tree has been reached.
if node.Leaf then
    return node.Hash;
// Recursive calls
leftHash := node.Left.verifyNode();
rightHash := node.Right.verifyNode();
return sha3.Sum256(append(leftHash, rightHash);
```

5.5.3 Leaf Verification

In the light client's mobile environment, rebuilding the merkle tree as in Subsection 5.5.2 for verifying only a subset of transactions - the relevant once - is not efficient, considering the fact that an arbitrary long number of transactions can be saved within a block. Thus, a mechanism for verifying individual transactions as in Example 3 shown must be implemented in an efficient way. The light client requests for a certain transaction and its corresponding block all intermediary nodes which are necessary to recalculate the merkle root from the network. The network node receiving the request must recalculate the block's merkle tree and filter the intermediates (Algorithm 3).

Algorithm 3: Filter intermediate nodes for leaf

```
Data: leaf
Result: intermediateNodes
currentNode = leaf;
currentParent = leaf.Parent;
for currentParent != nil do
    left := currentParent.Left;
    right := currentParent.Right;
    if currentNode.Hash == left.Hash then
        intermediateNodes = append(intermediateNodes, right.Hash, currentParent);
    else if currentNode.Hash == right.Hash then
        intermediateNodes = append(intermediateNodes, left.Hash, currentParent);
    currentNode = currentParent currentParent = currentParent.Parent
```

When done, the intermediates are sent back to the light client. The light client uses them to recalculate the merkle root. If the root's hash value equals the one saved in the block, the light client can be sure that...

- ...the transaction requested has been saved in the block.
- ...the transaction requested has not been altered after the block has been minted.
- ...no transaction has been removed or added after the block has been minted.

5.6 State Calculation

After having relevant transactions filtered and verified, the account's state calculation can be started and the function *getState(*account)* is called. A pointer to the account object is given. The state changes are written directly on the object (Algorithm 4).

Algorithm 4: Light client: Account state calculation

```
Data: acc
for all relevant blocks do
    // Collect the block reward.
    if block.Beneficiary == acc then
        acc.Balance += blockReward;
    // Process AccTx
    for all AccTx do
        accTx := requestTx();
        if accTx.PubKey == acc || block.Beneficiary == acc then
            verify(accTx);
            if accTx.pubKey == acc then
                acc.IsCreated() = true;
            if block.Beneficiary == acc then
                acc.Balance += accTx.Fee;
    // Process FundsTx
    for all FundsTx do
        fundsTx := requestTx();
        if fundsTx.From == acc || fundsTx.To == acc || block.Beneficiary == acc
         then
            verify(fundsTx);
        if fundsTx.From == acc then
            if !acc.IsRoot() then
                acc.Balance -= fundsTx.Amount;
                acc.Balance -= fundsTx.Fee;
            acc.TxCnt += 1;
        if fundsTx.To == acc then
            acc.Balance -= fundsTx.Amount;
        if block.Beneficiary == acc then
            acc.Balance += fundsTx.Fee;
    // Process Config
    for all ConfigTx do
        configTx := requestTx();
        if block.Beneficiary == acc then
            verify(configTx);
            acc.Balance += configTx.Fee;
        // Configuration parameters must be updated client-side.
        UpdateConfigParameters(configTx);
```

First, Algorithm 4 checks if the account belongs to the miner who minted the block. If this is the case, the account collects the block reward in *Bazo* coins (line 3). Then all *AccTx* of the corresponding block are processed. If the account's address equals the *AccTx*'s pubKey property, the account can be considered as created. There is one exception: The initial root account has no *AccTx*, since the initial root account's address is hard coded [18]. However, this does not concern new created root accounts. Since a root account can fund some other account without having coins substracted, this is relevant for state calculation (line 17). Afterwards, all *FundsTx* and *ConfigTx* are processed. For *FundsTx*, the Amount is either added or subtracted respectively. For all types of transactions it is required to first verify the transaction (lines 7, 15, 26) and to collect the transaction fee if the account is the block's beneficiary.

5.7 REST API

This Section describes the light client's REST API and its endpoints. The API implements *HTTP GET* and *HTTP POST* requests for account querying, transaction creation and sending respectively. For requesting data *HTTP GET* methods are used. Since *HTTP POST* requests are not cached and safer, methods submitting data are implemented using *HTTP POST* methods [22]. The API is available on port *8001* by default but can be changed in *p2p.go*. *REST.go* in the REST package initiates the interface and routes the incoming requests to the corresponding endpoints.

5.7.1 GET /account/:address

This endpoint is called for requesting account information. The following parameters are included in the request:

ADDRESS The account's public address (public key).

The light client answers with a JSON formatted reply:

```
{
    "code": 200,
    "content": [
        {
            "name": "account",
            "detail": {
                "address": "f894ba7a24c1[...]",
                                "balance": 1136,
                        "txCnt": 1,
                        "isCreated": true,
                        "isRoot": true
            }
        }
    ]
}
```

The endpoint accepts the hashed address as well. In this case, the light client first requests the account's address from the network by the hash value.

5.7.2 POST /createAccTx/:header/:fee/:issuer

This endpoint is called for creating an *AccTx*. The following parameters are included in the request:

HEADER The header for the transaction.

FEE The fee payed for the transaction.

ISSUER The root account issuing the transaction.

The light client answers with a JSON formatted reply:

```
{
    "code": 200,
    "message": "AccTx successfully created.",
    "content": [
        {
            "name": "PubKey1",
            "detail": "b8999ba5ce36[...]"
        },
        {
            "name": "PubKey2",
            "detail": "e8912b65384d[...]"
        },
        {
            "name": "PrivKey",
            "detail": "5d9dfe8d55b7[...]"
        },
        {
            "name": "TxHash",
            "detail": "39c6376b0b6d[...]"
        }
    ]
}
```

5.7.3 POST /sendAccTx/:hash/:signature

This endpoint is called for sending an *AccTx* to the network. The following parameters are included in the request:

HASH The transaction's hash value.

SIGNATURE The transaction's signature (Sig1).

The light client answers with a Code 200 if the transaction has been sent successfully.

5.7.4 POST /createFundsTx/:header/:amt/:fee/:txCnt/:from/:to

This endpoint is called for creating a *FundsTx*. The following parameters are included in the request:

HEADER The header for the transaction.

AMT The amount payed from sender to receiver.

FEE The fee payed for the transaction.

TXCNT The sender's transaction counter.

FROM The sender's account address.

TO The receiver's account address.

The light client answers with a JSON formatted reply:

```
{
    "code": 200,
    "message": "FundsTx successfully created.",
    "content": [
        {
            "name": "TxHash",
            "detail": "e96a86570482[...]"
        }
    ]
}
```

5.7.5 POST /sendFundsTx/:hash/:signature

This endpoint is called for sending a *FundsTx* to the network. The following parameters are included in the request:

HASH The transaction's hash value.

SIGNATURE The transaction's signature (Sig1).

The light client answers with a Code 200 if the transaction has been sent successfully.

5.7.6 POST /createConfigTx/:header/:id/:payload/:fee/:txCnt

This endpoint is called for creating a *ConfigTx*. The following parameters are included in the request:

HEADER The header for the transaction.

ID The configuration parameter's id.

PAYLOAD The new value to be set for the parameter.

FEE The fee payed for the transaction.

TXCNT The sender's transaction counter.

The light client answers with a JSON formatted reply:

```
{
    "code": 200,
    "message": "ConfigTx successfully created.",
    "content": [
        {
            "name": "TxHash",
            "detail": "90a95f2aaee4[...]"
        }
    ]
}
```

5.7.7 POST /sendConfigTx/:hash/:signature

This endpoint is called for sending a *ConfigTx* to the network. The following parameters are included in the request:

HASH The transaction's hash value.

SIGNATURE The transaction's signature (Sig1).

The light client answers with a Code 200 if the transaction has been sent successfully.

5.8 Multi-signature

The multi-signature server is a small Go program called *Bazo-multisig* within the *Bazo* project [4]. On startup, it listens on port *8002* by default for incoming *FundsTx* or updates to sign (Sig2) as described in Section 4.5. The port can be changed in *p2p.go* within the *bazo-client* program. It maintains open *FundsTx* within application memory. These are transactions which have been signed by the program but are not yet verified by the network. As soon as a *FundsTx* has been verified, the network sends an update to the multi-signature server and the open transaction gets deleted. Figure 5.4 explains how the multi-signature server decides if a transaction has to be signed or rejected.

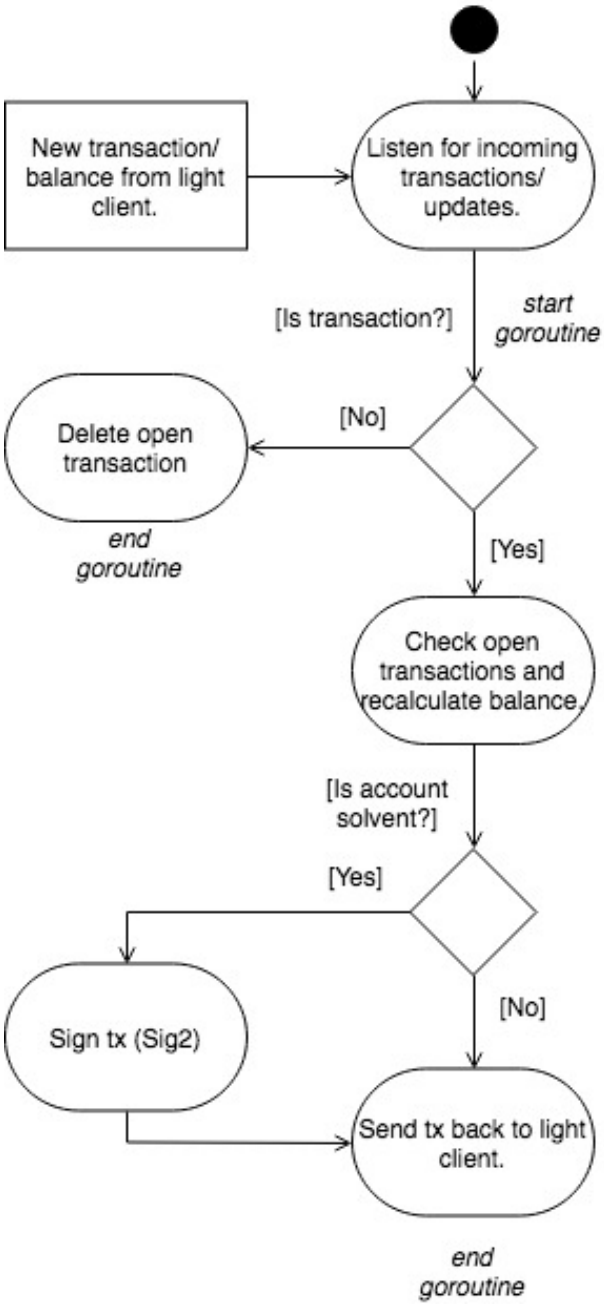

Figure 5.4: Multi-signature protocol activity

Chapter 6

Evaluation

The light client's requirements were defined in Chapter 3 as the following:

1. A mobile device must be able to calculate one or more accounts' states and display relevant account information such as the address and balance.
2. A mobile device must be able to send funds to the network in a reliable and secure way.
3. A mobile device must be able to verify the funds received within three seconds.

6.1 Running light client on mobile devices

The *bazo-client* application does not run on mobile platforms like *Android* on *iOS*. *Go mobile* provides support by either writing all-*Go* native mobile applications or writing *SDK* applications by generating bindings and invoke them from *Java* or *Objective-c* [11]. Nevertheless, type restrictions must be considered given by the platforms. For instance, the *Android*-platform supports only a subset of Go types (*e.g.* signed integer and floating point types) [12]. Unfortunately, these requirements can not be met considering the current *Bazo* version. Thus, running the light client on a mobile device is not yet possible. For the financial service provider's use case, a fallback solution has been planned and will be activated.

The fallback solution plans to run the light client on a trusted server. Instead of running the light client locally and serving the *REST* API over the local interface, the light client runs on a server owned by the financial service provider.

6.2 Downloading Block Headers

As stated in Section 4.4: 1 million block headers, resulting in 1 GB of size, should be downloaded within five minutes. This is with the current implementation not possible,

since *TCP* is used. Thus, each time before requesting a header, a new connection has to be established and afterwards closed again. This generates overhead resulting in latency. Table 6.1 shows downloading times for different amounts of block headers.

Number of block headers	**Duration** [mm:ss]
5'000	03:40
10'000	07:21
20'000	14:42
30'000	22:01
40'000	29:24
50'000	36:24
100'000	73:24

Table 6.1: Downloading times for different amounts of block headers

When all headers are downloaded, the state can be calculated very efficiently without any delay, since only relevant information has to be requested from the network and Requirement 1 can be satisfied instantly.

6.3 Sending FundsTx

Since the light client does not have to maintain sent transactions, no problems occur due to high work load and Requirement 2 is satisfied.

6.4 Verifying FundsTx

Implementing multi-signature mechanism allows to verify *FundsTx* even under three seconds. Thus, Requirement 3 is satisfied. As stated, the current implementation allows only multi-signed *FundsTx*. For the financial service provider's use case this is feasable. But this is subject to the following reservations:

- If the multi-signature server faults, no *FundsTx* can be processed anymore.
- Currently, only one multi-signature server can process *FundsTx*. Since the multi-signature server is not a trustless component, only the entity controlling the multi-signature server can reliably verify incoming *FundsTx*.
- The entity controlling the multi-signature server controls *FundsTx* processing. If a user sends funds to an account which is not controlled by the entity controlling the multi-signature server, this is a problem and could lead to malicious behaviour.

Chapter 7

Summary and Conclusions

Taking evaluations in Chapter 6 into consideration, Requirements 1-3 are satisfied partially. One or more accounts' states can be calculated while only downloading relevant data and account information can be displayed as required while the user's privacy can be kept safe. Transactions can be sent reliably to the network. *FundsTx* can be verified almost instantly due to multi-singature mechanism. But to cope with the trustless *Bazo* network, running the light client on the device directly is indispensable. Therefore, *Bazo* must be adapted in order to work also on mobile platforms. Further, the light client should persist downloaded block headers. This allows the light client to only download new block headers after being shut down. However, Section 7.1 describes future work derived from the conclusion.

7.1 Future Work

- **Light client running on mobile devices**
  It is important that the light client can be run locally on mobile devices while serving applications. Therefore, *Bazo* must be adapted as soon as possible to meet the corresponding requirements for mobile platforms evaluated in Chapter 6. Further should the light client's source code download and build be integrated into native mobile applications seamlessly. Otherwise, users without the required know-how are not capable of using the light client.

- **Downloading block headers**
  In order to avoid network overhead when opening or closing connections respectively, streaming sockets or *UDP* connections should be considered. Further should be discussed, if timepointing when downloading block headers is an option. Since all blocks before the one block where the account's *AccTx* is stored in are irrelevant, their download can be omitted. Thus, for each account the block must be known, where its *AccTx* is stored in.

- **Persisting block headers**
  Since blocks are immutable once they are stored in the blockchain, so are the block

headers. Thus, the light client can persist already downloaded block headers in permanent memory. On startup, only new block headers must be downloaded.

- **Implement simple- and multi-signature accounts**
  If the user receiving funds wants to verify the transaction immediatly, the receiver's account must be a multi-signature account. Thus, all incoming *FundsTx* must be multi-signed. The receiver must trust the multi-signature server that it pre-verifies the transaction correctly. If a receiver does not need to verify a transaction immediatly, the receiver only needs a single-signature account. Thus, the *FundsTx* must only be signed once by the sender.

Bibliography

[1] B. Akyildiz. "A Gentle Introduction to Bloom Filter." Internet: https://bugra.github.io/work/notes/2016-06-05/a-gentle-introduction-to-bloom-filter/, Jun. 5,2016 [Jan. 10,2018].

[2] B. Akyildiz. "Basic Math on How Bloom Filter Works." Internet: http://bugra.github.io/work/notes/2016-08-27/basic-math-on-how-bloom-filter-works/, Aug. 27,2016 [Jan. 10,2018].

[3] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. Sebastopol CA: O'Reilly, 2014.

[4] "Bazo-Blockchain project." Internet: https://github.com/bazo-blockchain, [March. 7,2018].

[5] "Bitcoin." Internet: https://bitcoin.org/en/, [Apr. 26,2017].

[6] "Bitcoin Developer Guide." Internet: https://bitcoin.org/en/developer-guide#block-chain, [Nov. 15,2017].

[7] "BitInfoCharts." Internet: http://bitinfocharts.com/, Nov. 15,2017 [Nov. 15,2017].

[8] D. Chaum. *Advances in Cryptology - Blind signatures for untraceable payments*. Boston MA: Springer, 1983, pp.199-203.

[9] "Cryptocurrency Market Capitalizations." Internet: https://coinmarketcap.com/, Nov. 15,2017 [Feb. 26,2017].

[10] "Ethereum." Internet: https://ethereum.org/, [Apr. 26,2107].

[11] "Go mobile." Internet: https://github.com/golang/go/wiki/Mobile, [March. 6,2018].

[12] "The Go programming language." Internet: https://golang.org/, Feb. 16,2018 [Feb. 22,2018].

[13] A. Lella, A. Lipsman. "2016 U.S. Cross-Platform Future in Focus." Internet: https://www.comscore.com/Insights/Presentations-and-Whitepapers/2016/2016-US-Cross-Platform-Future-in-Focus, March 30,2016 [Nov. 15,2017].

[14] R.C. Merkle. (1988). "A Digital Signature Based on a Conventional Encryption Function." *Advances in Cryptology - CRYPTO '87*. [Online]. 293(1), pp. 369-378. Available: https://link.springer.com/content/pdf/10.1007%2F3-540-48184-2_32.pdf [Jan. 10,2018].

[15] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". Internet: https://bitcoin.org/bitcoin.pdf, 2009 [Nov. 15,2017].

[16] OpenSignal. "The State of LTE (June 2017)." Internet: https://opensignal.com/reports/2017/06/state-of-lte, [Nov. 15,2017].

[17] M. Pilkington. (2015, Sept.). "Blockchain Technology: Principles and Applications." *Research Handbook on Digital Transformations*. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2662660, [Nov. 15,2017].

[18] L. Sgier. "Bazo - A Cryptocurrency from Scratch." M.A. thesis, University of Zurich, Zurich, 2017.

[19] D. Starobinski, A. Trachtenberg and S. Agarwal. (2003). "Efficient PDA synchronization." *IEEE Transactions on Mobile Computing*. [Online]. 2(1), pp. 40-51. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1195150 [Jan. 10,2018].

[20] B. Stiller, D. Hausheer, T. Bocek. "Distributed Hashtables with TomP2P." Internet: http://www.csg.uzh.ch/csg/dam/jcr:c45b8d8e-9657-45c5-8f95-fe37f889bec7/M04-1up.pdf, March 14,2017 [Jan. 10,2018].

[21] M. Swan. *Blockchain - Blueprint for a New Economy*. Sebastopol CA: O'Reilly, 2015, pp. viii-xi.

[22] "w3schools." Internet: https://www.w3schools.com/tags/ref_httpmethods.asp, [March 5,2018].

Abbreviations

AccTx	Account Creation Transaction
API	Application Programming Interface
CLI	Command-line Interface
ConfigTx	System Parameters Transaction
FundsTx	Fund Transferring Transaction
GB	Gigabyte
JSON	JavaScript Object Notation
KB	Kilobyte
Kb	Kilobit
MB	Megabyte
Mb	Megabit
Mbps	Megabit per Second
RAM	Random-access Memory
REST	Representational State Transfer
TCP	Transport Control Protocol
UDP	User Datagram Protocol

Glossary

Blockchain Blockchain is a continuously growing list of records, called blocks, which are linked and secured using cryptography.

Distributed Public Ledger Distributed Public Ledger is a consensus of replicated, shared, and synchronized digital data which is publicly available.

Ethereum Ethereum is a decentralized platform for applications that run exactly as programmed without any chance of fraud, censorship or third-party interference.

Full client A full client (also denoted as node) has a local copy of the blockchain. A full client can mine new blocks.

Light client A light client only keeps relevant information of the blockchain. A light client can not mine new blocks.

Public key encryption Public key encryption, in which a message is encrypted with a recipient's public key. The message cannot be decrypted by anyone who does not possess the matching private key.

Transaction In *Bazo*, a transaction denotes an *AccTx*, *FundsTx* or *ConfigTx* and is recorded in the distributed ledger. A transaction can change the network's state.

List of Figures

List of Tables

Appendix A

Installation Guidelines

The source code for all applications is in the public available *bazo-blockchain* repository on GitHub:

https://github.com/bazo-blockchain

Prerequisite
The programming language Go (developed and tested with version >= 1.9) must be installed, the properties $GOROOT and $GOPATH must be set. For convenience, add the $GOPATH to your PATH:.

A.1 Miner Application

If the miner application is started as *BOOTSTRAP* [18] for the first time, the initial root's public key must be set in the code in *storage.p2p.go*. For generating a keypair (public- and private key), see Guidelines A.4.1.

```
# Download the bazo-miner application.
go get github.com/bazo-blockchain/bazo-miner

# Run the application.
bazo-miner <dbname> <ipport> <validator> <multisig>
```

If no database is available under the given name, a new one is created. If the miner is not started as *BOOTSTRAP*, an empty database must be given. The *ipport* number must be prefixed with ":". The *validator* is the keyfile's name containing the validator's public key. The *multisig* is the keyfile's name containing the multi-signature server's public key.

A.2 Light Client Application

As described in Section 5.1, the light client can be started for different purposes.

```
# Download the bazo-client application.
go get github.com/bazo-blockchain/bazo-client
```

A.2.1 Sending an AccTx, FundsTx or ConfigTx

```
# Run the application for sending AccTx.
bazo-client accTx <header> <fee> <root> <new>
```

The *root* is the keyfile's name containing the root's public- and private keys. The *new* is the new users keyfile's name. It must not exist.

```
# Run the application for sending FundsTx.
bazo-client fundsTx <header> <amount> <fee> <txCnt> <from> <to> <multisig>
```

The *from* and *to* are the keyfiles' names containing the sender's or receiver's public- and private keys respectively. The *multisig* is the keyfile's name containing the multi-signature server's public- and private keys.

```
# Run the application for sending ConfigTx.
bazo-client configTx <header> <id> <payload> <fee> <txCnt> <root>
```

The *root* is the keyfile's name containing the root's public- and private keys.

A.2.2 Request Account State

```
# Run the application for requesting an account's state.
bazo-client <keyfile>
```

The *keyfile* is the keyfile's name containing the account's public key.

A.2.3 Start REST API

```
# Run the application for starting the light client's REST API.
bazo-client
```

No arguments must be given.

A.3 Multi-signature Application

```
# Download the bazo-multisig application.
go get github.com/bazo-blockchain/bazo-multisig

# Run the application.
bazo-multisig <multisig>
```

The *multisig* is the keyfile's name containing the multi-signature server's public- and private keys.

A.4 Utility Applications

These Guidelines contain instructions for utitlity applications.

A.4.1 Keypairgen

The *bazo-keypairgen* application generates an ECDSA curve P-256 keypair as required for *Bazo* [18].

```
# Download the bazo-keypairgen application.
go get github.com/bazo-blockchain/bazo-keypairgen

# Run the application.
bazo-rootgen <keyfile>
```

The application will output the public key's hash value and a new file under the name given in the current directory. The file contains the public- (first two lines) and the private key (third line).

A.4.2 Signtx

The *bazo-signtx* application is for testing purposes only. It allows the sign a given transaction's hash value with a private key.

```
# Download the bazo-signtx application.
go get github.com/bazo-blockchain/bazo-signtx

# Run the application.
bazo-signtx <txHash> <keyfile>
```

The application will output the signature.

Appendix B

Contents of the CD

The CD includes the following data:

- **bazo-blockchain** - The application's source code.
- **masterthesis.pdf** - The final version of the report.
- **midterm_presentation.ppt** - The midterm presentation.
- **related_work** - Related work paper (pdf or full html).
- **report** - LATEX source files.
- **report/images** - Image source files for report.