h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

HOCHSCHULE DARMSTADT

Fachbereich Informatik

Peer-to-peer gaming on the Bitcoin Blockchain

Abschlussarbeit zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

vorgelegt von:
Felix Schuchmann

Referent: Prof. Dr. Moore, Ronald Charles
Korreferent: M.Sc. Reuschling, Nicolai Constantin

Ausgabedatum: 05.04.2016
Abgabedatum: 05.10.2016

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit „Peer-to-peer gaming on the Bitcoin Blockchain“ selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Unterschrift:

_____

Darmstadt, den :

_____

HOCHSCHULE DARMSTADT
Fachbereich Informatik

Zusammenfassung

Master of Science (M.Sc.)

Peer-to-peer gaming on the Bitcoin Blockchain

von Felix Schuchmann

Inhalt dieser Arbeit ist die technische Konzeption und Entwicklung einer peer-to-peer Anwendung zwischen anonymen Personen mit finanzieller Motivation und ohne Vertrauensbasis. Die *Bitcoin Blockchain* Technologie wird als anonyme Kommunikationsebene benutzt und *smart contracts* als Grundlage für das Vertrauen auf finanzieller Ebene.

Als Machbarkeitsstudie ist ein kleines zwei Spieler *GO*-Spiel implementiert. Dieses Spiel veröffentlicht keine Identifikation der Spieler und benutzt zu keiner Zeit eine direkte Verbindung zwischen diesen. Gleichsam benutzt es keine zentralisierte Server-Instanz. Jegliche Kommunikation, für alle Aktionen in dem Spiel werden über das *Bitcoin* peer-to-peer Netzwerk abgewickelt. Die Gewinneinsätze zum Spielen werden über die crypto-Währung Bitcoin bezahlt und die möglichen Gewinne per *smart contract* ausgeschüttet. Dieses System ermöglicht völlige Anonymität der Spieler und verhindert gleichzeitig Engpässe, die beim Einsatz von einer Client-Server-Architektur anfallen würden. Zeitgleich eliminiert es die Notwendigkeit von Vertrauen in allen Bereichen.

Im Bereich dieser Arbeit werden diverse Methoden von verschiedenen, aktuellen *Blockchain*-Technologien, wie *Ethereum* oder *Lightning Network* besprochen. Diese Konzepte sind noch sehr jung und teilweise noch nicht umgesetzt. Deren Vor- und Nachteile kamen bei der Implementierung des *GO*-Spiels zum Tageslicht.

Es wird gesagt, dass die Zukunft dieser Technologie nicht nur die Fähigkeit hat, das Finanzsystem zu revolutionieren - wie man es *Bitcoin* nachsagt - sondern auch in anderen Gebieten zu einer digitalen Transformation führen kann.

HOCHSCHULE DARMSTADT
Fachbereich Informatik

Abstract

Master of Science (M.Sc.)

Peer-to-peer gaming on the Bitcoin Blockchain

by Felix Schuchmann

Content present in this thesis is the technical concept and development of a decentralized peer-to-peer application between anonymous persons with a financial motivation and no confidence base. The *Bitcoin Blockchain* technology is used as an anonymous communication layer with smart contracts handling the financial foundation of trust.

As proof of concept, a little game of *GO* for up to two players is implemented. The game does not reveal the identity of any player, nor does it ever use a direct connection between them, as well as a centralized server instance. Communication for every game action is made over the peer-to-peer Bitcoin network. The fees to play the game are paid with the *Bitcoin cryptocurrency* and the winnings are distributed with the use of *smart contract* systems. This guarantees absolute anonymity and avoids a bottlenecked server between many clients whilst eliminating the need of trust from the parts involved.

Within the scope of this thesis, different methods of current *Blockchain* based technologies, such as *Ethereum* and *Lightning Network*, are discussed. These, still very young or not yet readily implemented concepts, have their assets and drawbacks, which were revealed during the development of the *GO* game.

It is argued that the future lookout of this technology has the power to not only revolutionize the monetary system - as it is stated with Bitcoin - but also lead to a much bigger digital transformation.

Acknowledgements

I want to thank all those people, who made this thesis and my studies possible. I especially want to thank the whole JIM-team at the *Hochschule Darmstadt* and the international-team at the *James Cook University Australia*, who all made it possible for me to do my master degree with a abroad experience in Australia.

Additionally, I want to thank:

My girlfriend Jana Wieland, who supports my decision and helped me during stressful phases.

Philip Park who has helped me with some of the graphics.

Gabriel Lucas Cavalcanti for proofreading.

Mat Bell for his open source tool bitcoin-net.

Satoshi Nakamoto and every developer behind Bitcoin for this great invention.

"It's very attractive to the libertarian viewpoint if we can explain it properly.
I'm better with code than with words though."

Satoshi Nakamoto, 11/14/2008

Contents

List of Figures

List of Abbreviations

ASIC	**A**pplication-**s**pecific **i**ntegrated **c**ircuit
FPGA	**F**ield-**p**rogrammable **g**ate **a**rray
BTC	**Bit**coin as the currency, e.g. 0.5 BTC
ETH	**Eth**ereum as the currency
NMC	**Na**m**e**coin as the currency
RSA	**R**ivest, **S**hamir and **A**dleman
ECDSA	**E**lliptic **C**urve **D**igital **S**ignature **A**lgorithm
SegWit	**S**egregated **W**itness
UTXO	**U**nspent **T**ransaction **O**utput
MH/s	**M**ega**H**ashes per **s**econd
TH/s	**T**era**H**ashes per **s**econd
PH/s	**P**eta**H**ashes per **s**econd

List of Symbols

฿ bitcoin 1.00000000฿

Chapter 1

Introduction

Today, more and more business models are subject to change in the process of digital transformation. Many (business) processes shift from traditional, physical models to digital ones. Of these, many become automated and autonomous. Machines start to communicate and trade information with each other, but still, trust is a major factor for most of these centralized structures.

The Blockchain technology is a recurring topic on many current news outlets. It claims to solve the problem of establishing trust between two or more business partners while still maintaining the anonymity provided by the Internet. Smart contracts based on autonomous agents that replace a notary are finally not only a theoretical concept. But, what exactly lies behind this new technology? Can it be used in a practical manner? Would it possible for two anonymous persons to play a game against each other, where the rewards are a form of money, without trusting one another, or using a third party for the communication and safekeeping of their funds?

This thesis aims to answer those questions by implementing a simple game of *GO* based on Bitcoin and Blockchain technologies. Smart contracts will then handle the bets each player make.

1.1 Motivation

Whenever an online transaction has a financial background, be it an auction, gambling, providing a service, digital goods, or any other business interaction between anonymous parties, a major technical effort is necessary to maintain the network's trust and security. For

instance, when you sell a product to an individual over the Internet, you have to trust that the payment will not result in a charge-back, or that it does not originate from a compromised PayPal account, or even a stolen credit card. The buyer is often required an additional effort in order to verify their identity, either by uploading personal information, performing a postident, or in some cases, providing documentation directly to the seller. However, the merchant also has to be trusted with the delivery of the product. This relationship is generally ensured by a company charged with making the interaction possible, but a great deal of manual input is still necessary should a situation arise. The company is also entrusted with sensitive information such as credit card details, in turn making it a possible target for hacker attacks. Data leakage can compromise a company's image, potentially turning customers away due to distrust. From a technical perspective, there is no guarantee that an expanding software will be utterly secure.

The (Bitcoin) Blockchain allows data transmission over a decentralized network with thousands of peer-to-peer connected nodes. The protocol is open and comprehensible to everyone. Data can be transmitted from one person to thousands around the world without any centralized servers in between. Would it be possible to use this network for a project without having the well known drawbacks of centralized trust systems, servers, or current peer-to-peer networks?

1.2 Purpose

In this thesis, the possibility to use Blockchain – a relatively young technology – will be tested by developing a small peer-to-peer game for two players. A game is chosen as it presents a constant, bidirectional communication between two persons and the result is defined by a replicable goal.

The Bitcoin Blockchain currently only exists for its financial network, but many technology sectors research how to use if for their own applications. Plenty of these sectors have theoretical concepts, but not many, if any, practical working samples. One such sample is the resulting game of *GO* presented at the conclusion of this thesis.

1.3 Structure

In chapter 2 the basics of new Blockchain technologies are explained to give an understanding of the work that follows. Chapter 3 explains the game as a sample by using the techniques established on 2. 4 details implementation, how the application is structured, how communication happens within it (and with Blockchain), and how the smart contracts work. Finally, chapter 5 presents the project results as well as a conclusion and a further perspective on the matter.

Chapter 2

Basics

2.1 Important cryptographic basics

Although this thesis is meant for people experienced in information technology, I will explain some basics upon which the following techniques are based on. These are just crude explanations, but also the fundamentals to understanding the further structures.

2.1.1 Hashing

Hashing is a one way function that is often used in cryptographic context. Its goal is to take a string and mutate it into a non-reversible (usually shorter) string that represents the input. For example, one could take the first letter of each word in a sentence to generate this sentence's *hash*. When the sentence is sent over an unsafe communication channel and its *hash* over another, the receiver can check if the words in question still being with the same letters. If someone intercepting the message changes the sentence, they would have to make it match these letters. In detail, *hashes* are a lot more complex than the example and result in lesser collisions (generating the same *hash* twice with different input). They are generally used to save passwords for websites, so that if the website is hacked, no one could reconstruct the original password out of the *hash*.

In the next chapters, *hashes* are often based on the *sha256* and *ripmd160* algorithms. These *hashes* produce outputs of a fixed length of 256 or 160 characters and are popularly used and checked by many specialists for security purposes. They are considered safe.

2.1.2 Public key encryption

Public key encryption, or *asymmetric encryption*, is a widely used cryptographic procedure. A random number is used to generate a public and a private pair of keys. The private key is stored in a secret environment and used to decrypt or sign data. The public key is used to encrypt data or check the signatures for validation.

For this thesis, the signing part is more important than the encryption procedures. For instance, someone takes a message and uses their private key to generate a signature for the same. The data is sent with the unencrypted message to the receiver, who can use the public key to check with the inverse function the validity of the signature of the message, then claiming ownership over those.

Mostly *RSA* (named after its inventors Rivest, Shamir and Adelman [Mol02, p. 61]) is used as a well established asymmetric encryption. Its history goes back to 1974-1975 to Whitfied Diffie, Martin Hellman from Stanford and Ralph Merkle from Berkley [Mol02, p. 50]. But, for the cryptography behind Bitcoin, Satoshi Nakamoto chose to use *Elliptic Curve Digital Signature Algorithm* (*ECDSA*) over *RSA*. *ECDSA* compared to *RSA* uses less computation and data storage for its keys by maintaining the same security level [Sob+08, p. 488], making it the best choice for a peer to peer protocol that needs to store and transmit all data, as seen in the next chapters.

2.2 What is Bitcoin?

Bitcoin is a peer-to-peer electronic cash system founded by Satoshi Nakamoto in 2008. [Nak08] This system is designed to fit all fundamental properties of money and has many benefits over existing currencies. Those properties and the benefit of Bitcoin and its underlying technologies are explained in the next sections. The combination of this currency with its financial worth and the technology behind it brings possibilities that have never existed with any previous digital system.

2.2.1 Properties of money

Any mankind used monetary system needs to have some fundamental properties. Those are explained in the following, detailed list together with a comparison of Gold, common (paper) money and *cryptocurrencies* like Bitcoin. This should emphasize the benefits of this new sort of money as a basis for its later usage as a combination of economics and technical aspects.

Traits of Money	Gold	Fiat (US Dollar)	Crypto (Bitcoin)
Fungible *(Interchangeable)*	High	High	High
Non-Consumable	High	High	High
Portability	Moderate	High	High
Durable	High	Moderate	High
Highly Divisible	Moderate	Moderate	High
Secure *(Cannot be counterfeited)*	Moderate	Moderate	High
Easily Transactable	Low	High	High
Scarce *(Predictable Supply)*	Moderate	Low	High
Sovereign *(Government Issued)*	Low	High	Low
Decentralized	Low	Low	High
Smart *(Programmable)*	Low	Low	High

FIGURE 2.1: fundamental properties of money

Source: `http://www.maxkeiser.com/2016/07/how-bitcoin-is-slowly-replacing-fiat-currencies/`

- Fungible: A currency needs to be fungible, which means that it can be mutually replaced. For example, one ounce of any pure

element, like gold, is equivalent to any other existing ounce of the same element. Gold is fungible. One American Dollar or any other Fiat currency (from the latin, "let it become") can be exchanged to any equivalent American Dollar. The same way, one Bitcoin is also equivalent to any other Bitcoin.

- Non-consumable: One could exchange vegetables as a currency system, but once consumed they will cease to exist. All known and good monetary systems is non-consumable. So is Bitcoin. The tokens used in the Bitcoin system are only transmitted, not being consumed in any way.

- Portability: A currency needs to be easily transported. This becomes difficult with Gold due to its weight. Similarly, paper money becomes heavy when a big amount is stacked. Bitcoins are stored as a digital asset and can be ported on any digital storage device, or over any connection.

- Durable: Gold cannot be easily destroyed. Paper money can be lost forever relatively fast. 95% of all currencies nowadays are digital and can survive for as long as the system behind them exist. The same applied to Bitcoins. Once saved or backed up properly, they can survive the test of time, not being worn out or disappearing by itself.

- Highly Divisible: Here is one of Bitcoin's biggest benefits: it can be divided up to 8 digits behind the decimal point. As of today, the smallest amount of Bitcoin (0.00000001), also known as Satoshi, is worth $0.000005 EUR. This would be a unusable amount of gold and is also difficult to imagine as Euros. One Satoshi, on the other hand, can be used in a Bitcoin microtransaction as demonstrated later in the developed application.

- Secure: Counterfeits are an ever existing problem with physical currencies, something the underlying cryptographic functions of Bitcoin avert. The Bitcoin market capitalization is, now days, worth $9,5 billion USD [Coi16] and has proven over the last 8 years that it is secure enough not to be hacked, not even once. (Not to be mixed up with Bitcoin exchanges)

- Easy Transaction: It is difficult to store and transfer gold safely enough, as it only exists as a physical good. Digital currencies, such as Fiat and Bitcoin, can be exchanged between databases with no geographical relation. Peer-to-peer exchanges can be easily made with physical currencies, but also with Bitcoin if both parties have access to the Internet or exchange the private keys mentioned further ahead.

- Scarce: One of the biggest benefits of gold or Bitcoin over Fiat money issued by the government is the scarcity, as Fiat is a hyper inflation currency that can be newly created without limitations. Gold is finite, as well as hard to find and produce. Bitcoin has a limited supply of digital coins. There will be 21 million Bitcoins with a decreasing rate, generated in a process called *mining* (explained in chapter 2.3.4) similar to gold digging.

- Sovereign: The reason why Fiat money still surpasses any other type of currency is the government's involvement, which issues and backs it. Its volatility is expected low enough for it to be used as a daily form of money. This is one of Bitcoin's issues, as its market value fluctuates by 20% or more within minutes after the announcement of big news.

- Decentralized: The central control of gold or Fiat money is very high. The banks and government control the existence and all transferences in centralized vaults or databases. Bitcoin, as a major opponent, is controlled by the later explained cryptographic functions and the peer-to-peer network, with thousands of independent persons behind it. No single person or group can control this network. This will also be used as a main reason of choice to base the later application on. The code for Bitcoin's related software is open source and as of today, developed by 400 contributors. [Laa+16]

- Smart: Bitcoin is the first monetary system that can be programmed and used to create the so called *smart contracts*. These are contracts between two authorities that are handled only by the underlying technology instead of a third party. Smart contracts do not require one or more individuals to trust one another.

2.2.2 From Bitcoin to the Blockchain database

Now that we know why Bitcoin is this powerful when compared to other types of currency, this leads to bigger possibilities that the network behind it provides – and also why it can be used in projects beyond its financial aspect. All funds in the Bitcoin network are, as mentioned, stored in a decentralized database, the so called *Blockchain* (further explained in 2.4). The *Blockchain* can be used to store other types of data and the fact that the Bitcoin market cap is big guarantees that the data is stored permanently with the backup of 9,5 billion USD, also safe enough to be used for other high risk projects.

2.3 Technical background of Bitcoin

The later explained basics behind Bitcoin are based on cryptographic functions, mainly public and private keys used fo encryption/decryption, hashing functions and *proof of work*. With this well known mathematical methods, Satoshi Nakamoto created the first digital decentralized currency that could exist without the control of an exclusive authority, like other known currencies.

For this thesis it is important to understand the techniques behind Bitcoin and how the protocol works. This will be further explained in the next chapter.

2.3.1 Bitcoin wallets and addresses

When someone owns bitcoins, the currency is stored in the so called *Bitcoin wallet*. A wallet is a simple file or document that saves the access to one or more Bitcoin addresses. An address is a 26-35 character long string of numbers and letters, often depicted as a base58 string (for distinctiveness with no I, l, 0 and O) [Bit16a]. An address can be generated offline and for free out of 2^{160} possibilities.

There is a website [Wal16] that demonstrates this number. It lists (or better said generates on the fly) all possible addresses with 128 keys on one page. This results in a total of 90462569716653277674664832038037428010029347093027269048910283704311063667 5 pages, an amount of data no system could ever scan through.

All funds and transferences in the Bitcoin network are transmitted from and to different addresses in a so called transaction.

Each Bitcoin address consists of two cryptographic keys, a private and a public one. The private key is part of a users wallet and is used to sign transactions to other addresses. A hash of the public key is – alongside all transactions and the transmitted value – stored in a "public database", known as Blockchain.

What follows are two sample addresses:
- 1CHdqa9caqvFBc9vRUxeVZZPJLBJDugTjZ
- 13NGQBtxjHiVTCDx93ySYX1gF4si93FuZR

The first byte (the number 1) represents the network and is always 1 for Bitcoin, n or m for Testnet, and capital L for Litecoin (a clone of the Bitcoin system). The next 20 bytes are a *ripemd160(sha256(<public key>))* and the last 4 are the checksum with a *sha256(sha256(<network byte><previous 20 bytes>))*. This function obfuscates the public key in case there is any method to calculate the private key from it and also verifies its format, being, with the checksum, an easy way to approve the validity of the given address without the need of querying it in the Blockchain.

2.3.2 Multisig Addresses

Addresses can also be generated with the combination of several private keys. They are often called "XofY" addresses. For example, a 2of3 address is generated by three different private keys and can be signed using two of those. This can be used for a shared wallet with the ability to add more people to approve outgoing transactions. A company with 4 stakeholders could setup a 3of4 address. This address would have a single public key that could be used to receive funds for the company. If one stakeholder wishes to spend said funds, they need at least 2 other colleagues to approve the transaction. This can be, along other methods, used for the later explained *smart contracts*.

2.3.3 Transactions and nodes

Whenever funds are transfered, a user or system signs a *transaction* with their private key(s) and transmits it to the Bitcoin peer-to-peer network. This network consists of *Bitcoin (full) nodes* and *Bitcoin miners*. The *nodes* in question are later used for the communication in the developed application. A *Bitcoin full node* is a server that can be setup by anyone and has access to the whole data (every public key and every ever existing transaction) in the Bitcoin network. This data takes 80GB of disk space at the time of this thesis, with 5100 existing nodes [21.16a].

Each *node* creates a peer-to-peer connection to 8 other *nodes*, then listens and transmits all *transactions* and saves them temporarily in its memory, the so called *mempool*. It also verifies that the *transactions* are of a valid format and have the right funds. A spending *transaction* always has to be part of one or more incoming addresses and one or more outgoing ones. It also has to consume all incoming funds in the sum of all outgoing addresses.

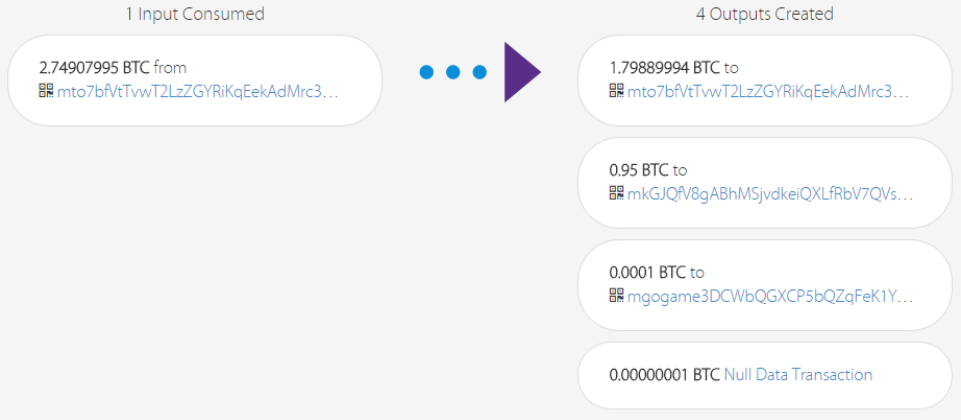


FIGURE 2.2: Example for a transaction

Source: `http://blockcypher.com`

Every Bitcoin transaction is written as a Forth-like scripting language [Ant14, p. 123]. Forth is a programming language from the 1970's developed by Charles H. Moore, using a stack based memory system and a *reverse-polish notation*. For instance, the calculation "2 + 3 = 5" would be written "2 3 + 5 =". Every time a *word* (+-*/=) is found in the dictionary, the function behind it is called and executes the parameters it has read until that point as input parameters.

If Bitcoin words in that dictionary are, for example, OP_ADD and OP_EQUAL, the resulting script from above would be "2 3 OP_ADD 5 OP_EQUAL", thus true.

Those scripts are stateless, meaning that all Bitcoin nodes have full access to them. They can be read as a combination of locking and unlocking parts, executed without any external information (stateless) and finally, verify that the result is true.

There lies one difference from Bitcoin to the later explained Ethereum coin: Bitcoin transactions are Touring incomplete because they do not implement loops or complex flow capabilities, while Ethereums added complexity do (as later seen in chapter 2.4.6).

Currently, the Bitcoin "livenet"-nodes only relay so called *standard transactions* that only allow five different transaction types: Pay-to-Public-Key-Hash (P2PKH), Public-Key, Multi-Signature (limited to 15 keys), Pay-to-Script-Hash (P2SH), and Data Output (OP_RETURN) [Ant14, p. 128].

The most interesting transaction types are Pay-to-Public-Key-Hash and the Data Output for this thesis.

The Pay-to-Public-Key-Hash transaction output looks like:

<signature> <Public Key>
OP_DUP OP_HASH160 <Receiver Key Hash> OP_EQUAL OP_CHECKSIG

The first row defines the script unlocking part that needs to be set correctly to spend that transaction. The second row defines the method used to fulfill that script together with the Bitcoin address. This part of the script is called *unspent transaction output* or *UTXO* and defines the funds in a user's wallet. To be spent, the first part of that formula needs to be fulfilled.

If a user wants to spend this *UTXO*, they sign the new transaction output and the amount of bitcoins with their private key, generating the signature. Then they add their full public key to that script for further verification. The cryptography behind Bitcoin can then verify that the sender is the legitimate creator of this transaction and can check the signature against the previous key hash, then match the new value with the previous transaction.

OP_RETURN is the second transaction type that is required later on. This is just a null data transaction which adds the possibility to add custom data to the transaction. Currently, this is limited to 83 bytes.

Each transaction can combine one or many different scripts, making it possible to send one transaction to many different destinations at the same time, also adding one OP_RETURN type to a transaction that transmits a value of bitcoins simultaneously.

2.3.4 Mining

Mathematically, every ten minutes a *miner* packs all transactions from their nodes' *mempool* into a *block*. This is done by creating a *sha265(sha265())* hash value of all transactions in this block plus the hash of the last existing block and a nonce. The nonce is a random number which the miner has to guess by using brute force techniques so that the resulting hash starts with a given and adjustable number of zeroes. Every 2016 blocks (or two weeks), the network adjusts the number of zeroes to match the computing power in order to maintain the ten minute interval.

Part of the protocol is also that the miner is allowed to generate one new transaction (Coinbase transaction) with no input address and an output value of X฿ , with X being initially 50฿ later halved every 210,000 blocks (or four years), today worth 12,5฿ (or 6500 EUR). This reward guarantees that enough miners try to solve this mathematical brute force problem as a *proof of work*.

The big profit from mining with the rising value of Bitcoin lead to a hardware race (specific for this end) which started in 2012 with the mass using of graphic processing units (GPUs). Many private people all around the world started to build mining rigs with up to 6 GPUs on one mainboard (see Figure 2.3), resulting in a powerful machine that only calculated sha256 hashes by using the GPUs' parallel shader architecture. At that time it was difficult to buy AMD GPUs, as the most power efficient models were sold out immediately to *miners*. AMD was chosen because it uses a different architecture than Nvidia, with more shader units running at a lower clock speed.

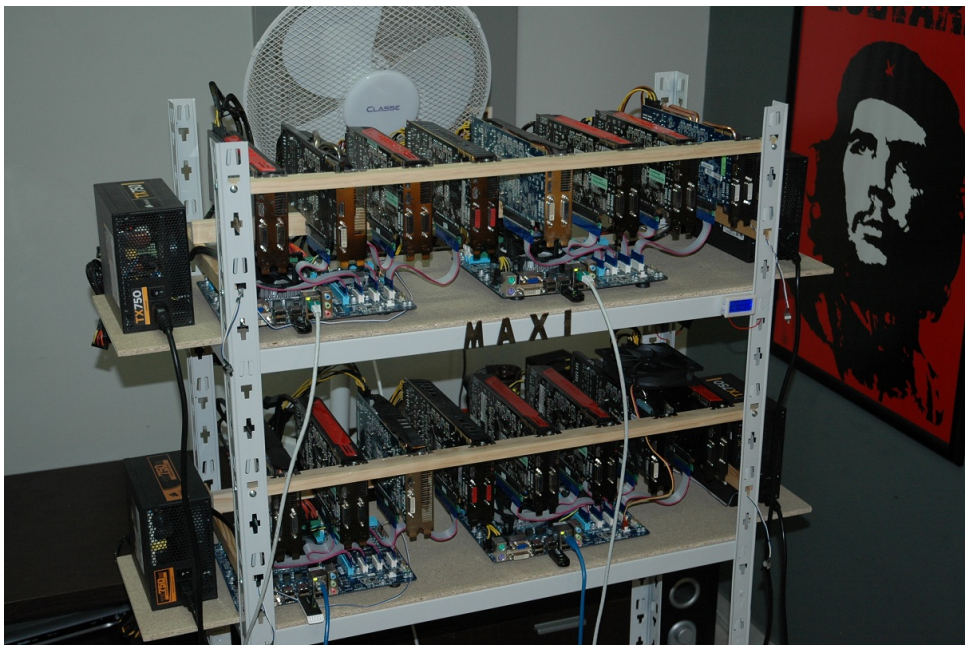


FIGURE 2.3: GPU mining RIG

Source: http://blogerator.ru/page/bitcoin-chast-2-kriptovaljuta-majning-zarabotok

Later on, companies started to reprogram FPGA processors, then designing special ASIC chips to optimize the process of hashing. Today, there are high specialized 14nm ASIC chips (e.g. from Bitmaintech [Bit16e]) that are capable of generating 8.6TH/s with the same power consumption of one of the older GPU rigs that only generated under 1000 MH/s per GPU.

The Bitcoin network brute force power is, today, 1800 PHashe/s (see Fig 2.4). Since no one can control so much computing power by themselves, the network is decentralized all around the world, with different private people and companies involved in mining.

2.3.5 Summary of the Bitcoin workflow

A user uses their Bitcoin wallet to hold their balance in form of addresses and their UTXOs. Then, they send transactions by signing the UTXO with the use of public key encryption to full nodes, addressed to a receiving Bitcoin address. Those nodes verify all transactions and relay them to other nodes in a peer-to-peer network. On top of each node there may be a Bitcoin miner who packs all transactions from the last ten minutes to a block by using a massive *proof*

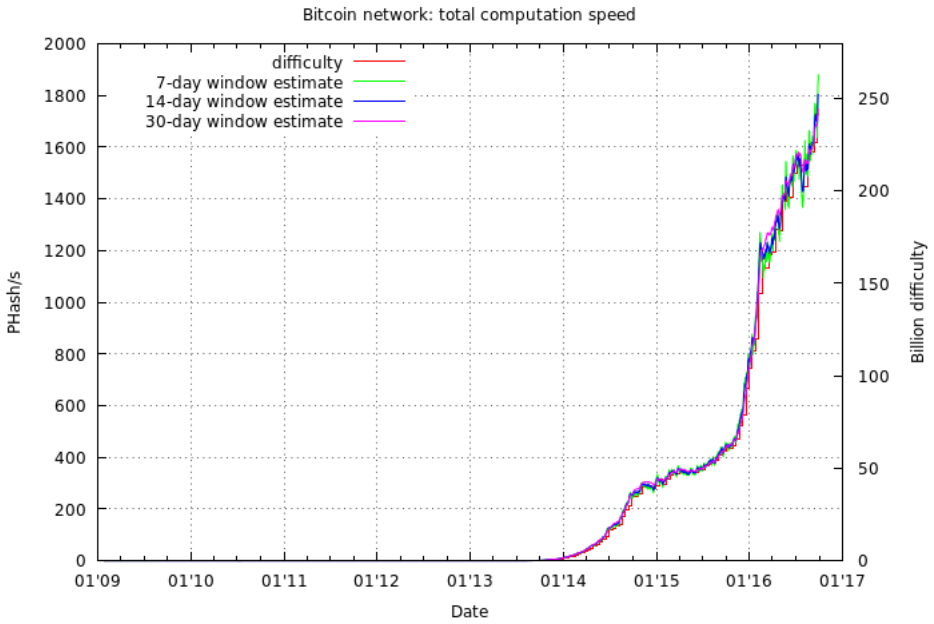


FIGURE 2.4: Graph of the total hashrate over time

Source: `http://bitcoin.sipa.be/k`

of work algorithm. This block is then again relayed to all other nodes. All blocks together form the Blockchain, Bitcoin's 80GB database that is saved on each full node's hard disk.

2.4 What is the Blockchain?

The Blockchain could be called Bitcoin's database. It is a chain (*merkle tree*) of all blocks that are mined, starting from the *genesis block*, the first ever block created by Satoshi Nakamoto.

Because every new block has a hash of the last one, a *chain* is arising. Whenever two miners generate a block at the same time, or a miner generates a block with invalid transactions (may it be faked transactions by the miner itself or transactions that already have been spent), the "chain" forkes to a tree-like structure and the miners choose what branch they will carry on mining. Being that anyone can verify the data in the block to be valid or not, the majority of miners would choose the valid one. When the next block is then found, the longest branch or chain wins the race. The transactions that are bound to the orphaned block go back to the *mempool* and are included in a later block. With the amount of work that is put

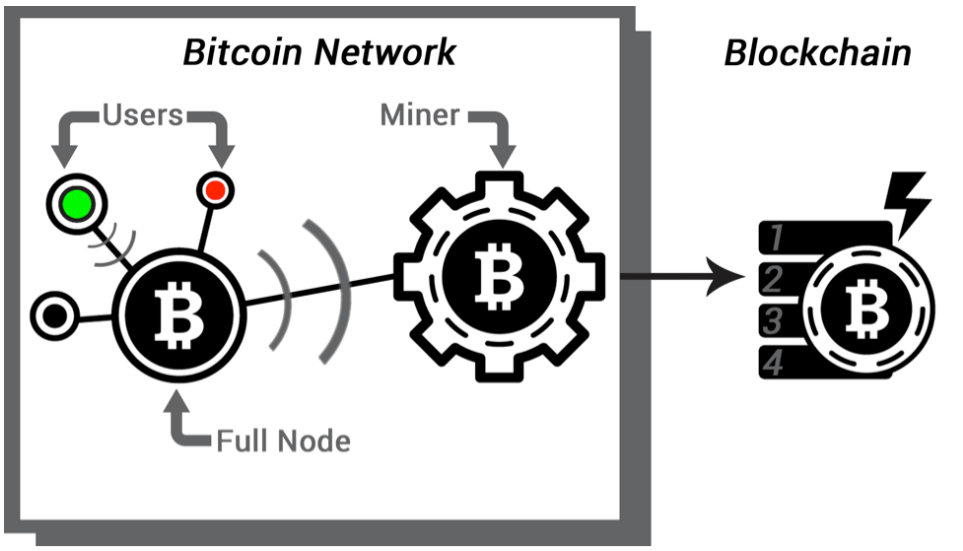


FIGURE 2.5: Bitcoin Network

Source: https://www.buybitcoinworldwide.com/kb/what-is-bitcoin-mining/

into verifying and generating the blocks, it is guaranteed that a transaction is irreversible and forever valid after some correctly chained blocks. This is defined as true after 6 blocks.

2.4.1 Data in the Blockchain

A special feature lays in the definition of transactions that are stored in the Blockchain. Each transaction can hold a *script opcode* called OP_RETURN that gives the ability to pack data after this return statement. The default node relays transactions with up to 83 bytes of data after the OP_RETURN. This makes the Blockchain usable as a public decentralized database, which is - beside its financial value - one of the most interesting things in the world of Bitcoin.

One example for using this Blockchain database could be a public *digital rights management* (DRM) system: Whenever a media creator wants to publish a photo, video or any other document they have created on their own, they could generate a hash over this data and transmits a transaction to the Bitcoin Network just before the publishing of the document. If another person wishes to prove that they are the actual owner of this document, they could simply look up the creator's public address and the outgoing OP_RETURN transaction

with the hash of the document. Using the unchangeable and decentralized Blockchain for the *proof of ownership* guarantees that no centralized organization could manipulate the data. In the same effect it is very cheap - if not even free - and easy to use this concept.

Today, many companies out of the financial (fintech), insurance, or even energy sector try to use the Blockchain and especially smart contracts over the Blockchain for similar processes.

2.4.2 Different types of Blockchains/altcoins

Due to the source-code of Bitcoin being publicly available, anyone can copy and modify the system for their personal use. One of the first such copies was the alternative coin (*altcoin*) named *Litecoin*. Its hashing algorithm is based on *scrypt* instead of Bitcoin's sha256. *Scrypt* is a memory intense hashing algorithm that was developed to cause brute force on special FPGAs or ASIC chips impossible. It is used in Litecoin to decentralize the process of mining to normal PCs instead of specialized companies with high developed ASIC chips.

Today, there are hundreds of altcoins with many different protocols and approaches. But the biggest network by far is still Bitcoin.

If someone decides to use the Blockchain as a storage or transmitting protocol, there are many more possibilities besides Bitcoin. The next sections describe the differences of open and closed Blockchains.

2.4.3 Closed Blockchain

A closed or private Blockchain can be fully or half private. For example, there could be public read permissions, but only a set of people or a single company could write data to the Blockchain. This could be done with a pre-mined amount of coins that is under the company's control or a special logic that only allows private miners to add blocks to this Blockchain.

Many fintech companies are today researching what can be done and how to benefit from this technology, for instance, to handle the stock market. Typical problems like race conditions in the balance of a wallet (double spents) that require locking mechanisms, if multiple instances try to access the same data, do not exist in the Blockchain because every transaction is unique, consumes all inputs and is verified by the mining mechanisms.

Some benefits of this closed Blockchains are:

- The person or company that has control over this Blockchain can revert or correct transactions, modify or block balances and change other rules by adjusting the source code or implementing other control systems. This is desired for many systems controlled by the government.
- The miners can be private and known, so no one can control the majority of the mining and perform a 51% attack. This type of attack defines that anyone who has 51% of the mining power can choose what chain will become the longest branch so that it can define what transactions are included to that chain, thus manipulating its content and allowing double spends and ultimately doing whatever is against the defined rules.
- Given the last benefit, it makes transactions much cheaper. There would be no hardware race to have the best hashing power and the Blockchain could be theoretically mined by a single unit.
- They can be faster as public Blockchains due to the known nodes and miners. The latency between the different servers can be optimized and the block time can be reduced to seconds instead of Bitcoin's ten minutes.
- The level of privacy is greater because the company can control who has access to what data.

One could say that if freedom, openness and neutrality are of no concern, then the private Blockchain is the finest tool.

2.4.4 Open Blockchain

The open or public Blockchain is the known system used by Bitcoin. It is readable and writable by anyone around the world, as well as completely decentralized. The cryptoeconomics (the combination of cryptographic features and the economic value behind Bitcoin) makes it one of the most secured database systems today.

On the other hand, no one can control or log who has access to the data. Such information could be required by some types of applications.

2.4.5 Namecoin

Namecoin is a good example of what can be done with an alternative Blockchain. What Namecoin does is basically a decentralized DNS system based on the Blockchain technology. To register a new .bit domain or browse the existing domains, the user needs to download the Namecoin client. Similarly to the Bitcoin client, it functions as a peer-to-peer node, downloading the Namecoin Blockchain to the local system. A middleware software is then used to connect a web browser to the Namecoin client and provide the lookup for .bit domains in the Blockchain. Whenever someone wants to register a domain they need the altcoin currency NMC and perform a transaction into the Blockchain with the domain name as the data payload.

The benefit of this over existing domain registrars, DNS servers and even HTTPS CAs is that Namecoin domains are censor resistant. No one can prevent anyone from registering a domain, just like no one can prevent someone from spending their bitcoins.

The local lookup system adds additional privacy and security measures to the whole domain. It is not possible to hijack .bit domains with man in the middle attacks or compromised DNS servers. The local system does not perform any DNS lookup at a central authority for every domain name the user looks up. These lookups were used to spy on users in the past, like 2014 in Turkey[1]. Also, .bit lookups are faster than standard DNS lookups and are updated much faster while still being much cheaper without the recurring registration costs.

2.4.6 Ethereum

Ethereum is one of the latest cryptocurrency projects and to date the second largest regarding the market's capitalization. While Bitcoin provides a method to script code into transactions (see 2.3.4), it lacks many required functions to write complete applications on top of the

[1] http://derstandard.at/1395363675874/Internetspionage-Tuerkei-leitet-Google-DNS-um

Blockchain protocol. For example, it is not possible to program loops into a transaction script. Ethereum enables this and other functions to make it "a decentralised secure social operating system" [Woo14, p. 14] or "a Blockchain with a built-in Touring-complete programming language" how it is named in its Wiki.

Ethereum briefly adds *contract code* and *storage* fields to all accounts. The code then gets executed like an *autonomous agent* whenever the account receives a transaction and consumes *gas*. *Gas* is a combination of the Ether currency and an execution cost for the script, used to prevent denial of service attacks. The agent then can interact with the storage, the accounts balance, or send messages to other agents.

With the given complexity, Ethereum made it possible to develop a *decentralized autonomous organization* (DAO), an institution without a conventional management or leadership. All decisions in the DAO are made by members who deposited Ethereum to it and attend to votes through the Blockchain.

Because this adds an additional layer of complexity on top of the young Blockchain technology, some people say that Ethereum is too complicated to be secure enough. To sustain that, the DAO was hacked on June 17th, 2016, and the counter-value of 45 million euros was stolen. The hack lead to an agreement among all developers to revert the hack-transaction in the codebase, but not all nodes and miners agreed with the decision. This resulted on a hard fork of the Ethereum currency with two existing Blockchains. The reverted ETH and the Ethereum Classic (ETC) fork with the existing 3,6 million ETC hack.

2.5 Smart contracts

Smart contracts are computer protocols that handle or prove contracts with technical methods. This can render the use of physical contracts unnecessary. They function autonomously and leave out some middlemen that need to verify a contract, like a notary.

Maiborn Wolff, a consulting company in Frankfurt, Germany, explains smart contracts to customers with the model in figure 2.6. Both trading partners place their property into one side of the tubes. Then,

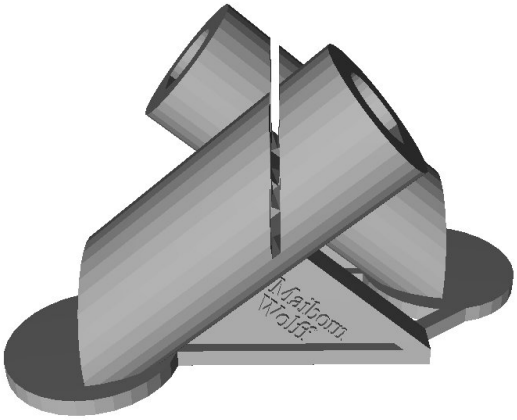


FIGURE 2.6: physical model to explain smart contracts from Maiborn Wolff

Source: https://twitter.com/droeder72/status/775229713157685248/photo/1

the agent who handles the contract is depicted as a slice in the middle. This first deny the properties from sliding to the other person's tube exit. When the agent verifies that both partners entered the right property or value, the gates are opened and the contents pass to the other person's side. Those persons do not have to trust each other or even need to know who they are trading with. The only thing they're required to do is verify the agent's source code or functionality.

The concept of smart contracts dates back to the 70's through the 80's [Hub88] with the goal to handle auctions and resource management with software. In the next thirty centuries, a lot of different concepts evolved on ways to handle contracts without a third party. Plenty of it was revolutionized with the public key cryptography (see 2.1.2), which is also used in Bitcoin and Ethereum, or other cryptocurrency concepts.

Next, there are some practical examples demonstrating what smart contracts can be used for.

2.5.1 Practical example 1: .bit domain in Namecoin

One example of a smart contract would be to sell the ownership for a .bit domain in Namecoin. In the classic model, buyer and seller need to trust each other when they transfer a domain. Whenever this trust model goes wrong, a legal prosecution follows and the funds or domain rights have to be transfered through legislative force. Given the help of a smart contract, built with software, there is no more need for trust. The seller can input the ownership of their domain on the contract. Only when the buyer inputs their ownership of the desired financial value (BTC, NMC, ETH, ...), the autonomous agent in the contract code automatically transfers the two values between the two parties.

2.5.2 Practical example 2: lent out photo equipment

Another practical example would be the possibility to lend out photo equipment to another person. The owner (individual A) would take a photo that proves the ownership with a personal ID, a daily newspaper with the current date, and the serial number of the equipment. They would then generate a digital hash for this photo and save it with the serial number to the Blockchain. Whenever someone (individual B) wants to rent this equipment, they would send the countervalue represented in any cryptocurrency to a smart contract. This smart contract then would execute an agent that transfers the previous saved hash, what illustrates the ownership, automatically after a predefined time frame from individual A to B, but also transfers the funds that are frozen to the smart contract to individual A. When individual B returns the rented equipment, they both agree to close the smart contract by calling a close function, which just transfers the ownership back to A and the funds back to B.

One problem would still remain with such a contract:

- The currencies volatility: The existing open cryptocurrencies underlay strong volatilities. After the defined amount of time, the countervalue in Euros could differ strongly from the equipment's real value. This can be handled by depositing a value that is higher to the contract. The contract then can call an *oracle* that replies with the current share price, calculates the amount of crypto coins to send to individual A and B and executes the

transference. This oracle is a service that saves the connection between the smart contract and any data API. One example of such service would be to call an API in given intervals and push the exchange rate data to the Blockchain. The code in the smart contract can then access this exchange rate data with a replicable outcome.

2.5.3 Nowadays interest in smart contracts

Today there is a big interest in many existing business domains to solve problems with smart contracts. May it be the stock market or even the autonomous machine to machine charging of an electric car on a power socket of public domain. The current state of smart contracts with Blockchain technology is still in very early phases and many ideas only exist as concepts. But there is a real interest in times of digital transformation. One example is the company Maiborn Wolff (Fig 2.6), who already provide consulting services with Blockchain technologies for some big companies like BMW [2].

In this thesis, the practical possibility of transferring the moves and funds for a competition in the game *GO* will be investigated and implemented as a proof of concept for smart contracts. This is a practical example of two persons having a financial motivation but no trust relationship.

[2] https://www.maibornwolff.de/blockchain

Chapter 3

Game presentation and user flow

This chapter handles the game presentation and the user engagement flow in the implemented game. First, the game *GO* is briefly described and then the different phases of the game are depicted with their implementations in form of the introduced Blockchain technology.

3.1 A game of GO

For this thesis, the basic board game *GO* is chosen. While it is not important to understand the rules of *GO*, since the focus of this thesis is defined by its technical aspects, they are much easier than that of chess, for example. In *GO*, two players (black and white) strategically play against each other. On each turn the player may choose to either move or pass. Within a move they place their respective stones in an alternating order on the game board. The size of the board varies, being either 19x19, 13x13, or just 9x9, for beginners.

The stones are not moved around like those of chess, making the technical communication aspect of it much easier to deal with. Each turn requires that the player either passes or plays by using their respective ID or color and the X/Y coordinated of the move. After every move, the rules are applied and stones may or may not be removed from the field. That happens when a stone or a group of the same has no *liberty*, an event that happens when they are surrounded by opposing stones or on the limits of the board with no space left to extend the group. Technically, that rule applies automatically on both clients and is not needed to communicate between them in any special form.

A fascinating aspect of *GO* is its complexity when compared to other strategic board games. The first supercomputer that defeated a professional *GO* player was Googles *AlphaGO* in March 2016 [Lim16]. The math behind the complexity of *GO* can be handled by a thesis alone, but as an interesting comparison, the amount of possible Bitcoin addresses is 2^{160}, while the amount of legal *GO* positions is $2.08168199382 * 10^{170}$ [Tro16].

This makes an online game of *GO* very resistant to players who try to cheat by using a computer program, resulting in an interesting example for the implementation with a smart contract winning system.

The game in this thesis is handled by the open source library *tenuki.js* by Adam Prescott [Pre16], which already has a client-server architecture built in. The server part is replaced by using the Bitcoin Blockchain protocol. Each client or player communicates through their browser with a random Bitcoin node by listening and writing directly from or to the Blockchain.

The developed game, its techniques and protocols are described in the following sections.

3.1.1 Visiting the website

First, when any user visits the website a lot of code is executed in the browser, not visible in the frontend. The user's wallet is generated and saved or just loaded if they are returning to the page. Then, the backend establishes a connection to some Bitcoin full nodes and the implemented services start to scan the Blockchain for games whilst listening for incoming Bitcoin transactions.

Visible for the user is the process where their wallet is created in the later explained walletService 4.3.2. They are then greeted by their own Bitcoin address, representing their identity in later games and also serving as a payment method for their wallets. The address is showed in Fig 3.1 as an input field, as well as a QR code. This code represents the address, making it easy for the Bitcoin wallets on smartphones to scan it. In the first implementation of the game this random address is fixed and stored in the browser's localStorage database.

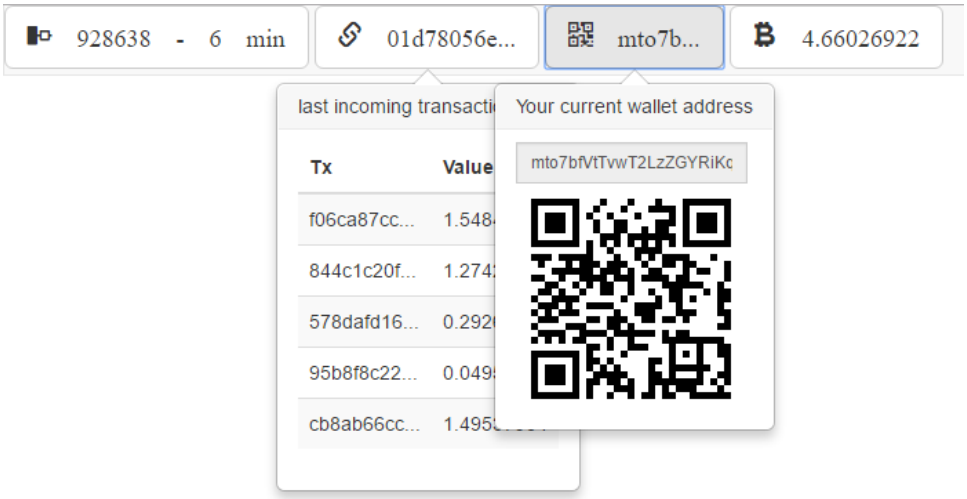


FIGURE 3.1: Game menu with open address and transactions list

Later on this should become extended to a whole wallet with the possibility to generate more incoming addresses and a method to withdraw funds. Normally, a Bitcoin address is used only once for privacy and security reasons, but can be used as many times as one wishes to. As an improvement, this wallet should be in the form of a *hierarchical deterministic* (HD) wallet as introduced in BIP32 by Pieter Wuille[Wui12]. The latter does not generate random addresses like this game does in its current version.

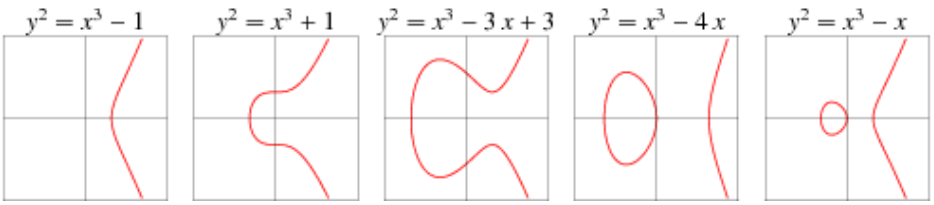


FIGURE 3.2: Elliptic Curve samples

Source: Wolfram Alpha

An HD wallet creates a random 128 bit seed that can be displayed with a 12 word mnemonic using a defined dictionary with common English words. These words are shown to the user, allowing them to make a backup of their wallet/account by saving those words. The wallet service then can derive new addresses with this starting seed by using elliptic curve cryptography. In short, this method takes the seed as a starting point to generate an elliptic curve like in 3.2

where each new address stands for a fixed X and Y coordinates in this curve. This process of generating and backing up a wallet is common in many Bitcoin projects and well known for the most users, experienced with Bitcoin.

3.1.2 Game start

After the user sets up their wallet and charges it with funds, the game can be started. They may either choose to host a new game or join an existing one. The screen in fig 3.3 shows the actions necessary to start a new game to the left and the options to filter existing games to the right. Below is a list of all existing games that match the defined filters.

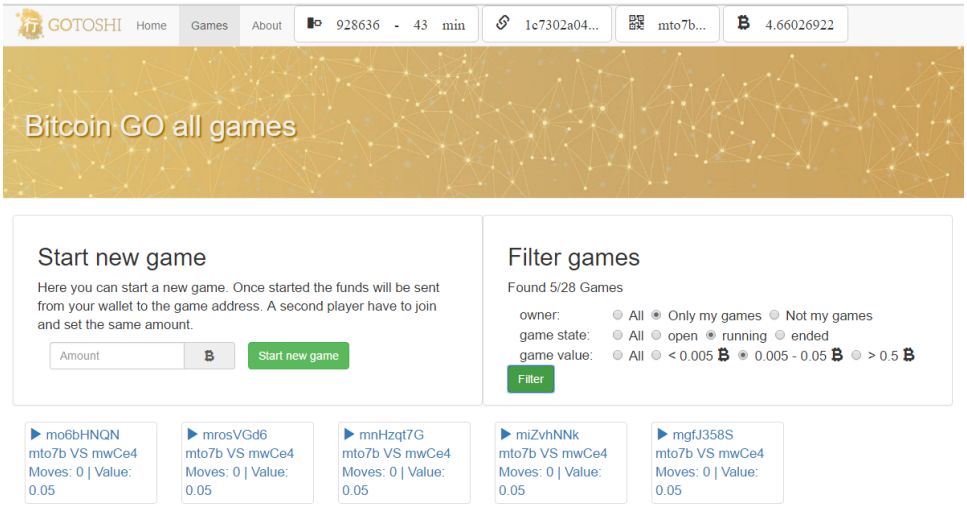


FIGURE 3.3: Game screen with new game and filters

If they wish to start a new game, the desired amount of Bitcoins must be set towards the winning pool before clicking the left button. The backend code then creates a transaction to the fixed *Bitcoin-GO-Game-address mgogame...* as seen in fig 3.4, with an OP_RETURN message (Output 4: Null Data Transaction in fig 3.4) that specifies the start of a new game). The transaction also submits the amount of winnable Bitcoins to a second, random generated address (Output 2: 0.95 BTC in fig 3.4), which defines the game and its later moves.

The sample transaction in fig 3.4 has one input and 4 outputs:

- First, the input address itself for the remaining balance.
- The random generated address for the game.
- Then, the *mgogame...* address where all clients listen to for the process of finding new games.
- And finally, output the OP_RETURN containing the *NEW-game* statement.

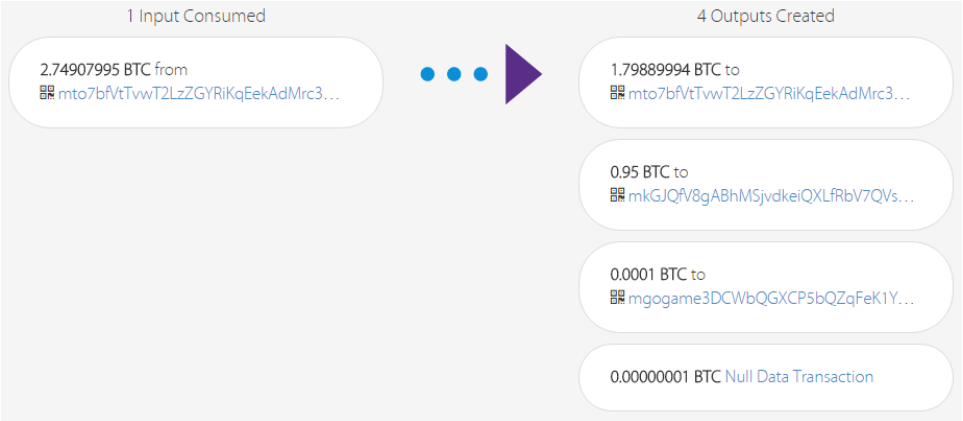


FIGURE 3.4: Sample start game transaction

Source: `https://www.blockcypher.com/`

Other players can later scan the Blockchain while listening to that fixed *mgogame...* address and find the OP_RETURN *NEW-game* statement. At the same time they see the outgoing Bitcoins to the random generated game address and can then sign up to play against the opponent. To achieve this they have to click *join*, allowing the game service to send the same amount from the user's private wallet to a new multisig address. It also generates a *JOIN-game* transaction to the fixed *mgogame...* addess to let the opponent know that the game is ready to start. The opponent then responds to this by also moving their bet to the multisig address. This process is later described in chapter 4.4, with the implemented smart contract in more detail.

Now both players are ready to start and have paid their stake to the multisig address. The game knows which address is used for the current instance and that both players have the permission to play.

As seen in the sequence diagram (fig 3.5), players one and two never have a direct connection to each other, maintaining anonymity.

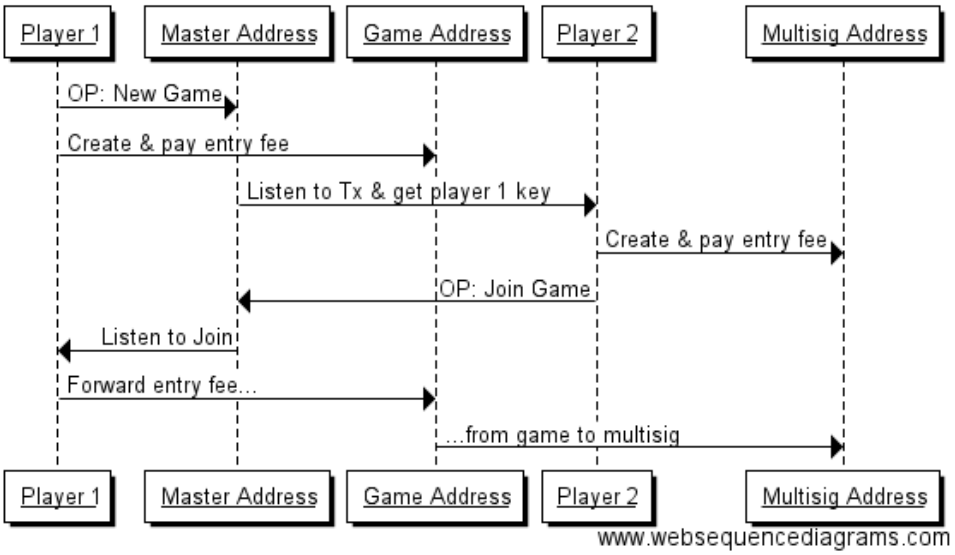


FIGURE 3.5: Sequence diagram for start and join game

Source: `https://www.websequencediagrams.com/`

3.1.3 Transmitting moves

To play a move, one player clicks on the board to set their stone. Fig 3.6 shows an ongoing game between two players with 13 moves.

For each move the gaming backend sends a transaction with an OP_RETURN code containing the X/Y coordinates of the move. Because of the alternating order of the players' and the origin address, it is not required to submit a color for the move. As a drawback of the Blockchain protocol, each move requires a small fee to the miners as a bounty for including this transaction to blocks. With each move this fee is subtracted from the player's private wallet.

As seen in 3.7, one move only has the input wallet and the unique game address as outputs. Also, the transferred value is only 2 Satoshis plus an added fee that allows this transaction to be included into the Blockchain. Such a transaction is roundabout 270 bytes large and needs a fee of 60 Satoshis [21.16b] for no delay. This would result in a cost of 8 cents for each move. Because it is enough to let the nodes reply that transaction and hold it into their mempool, the mining fee can be made much smaller. The game does not rely on fast confirmations from the miners for the moves. If we were to choose 10 Satoshis per byte, this would result in 1 cent per move and an estimate of 3 hours of confirmation time. This would only delay the

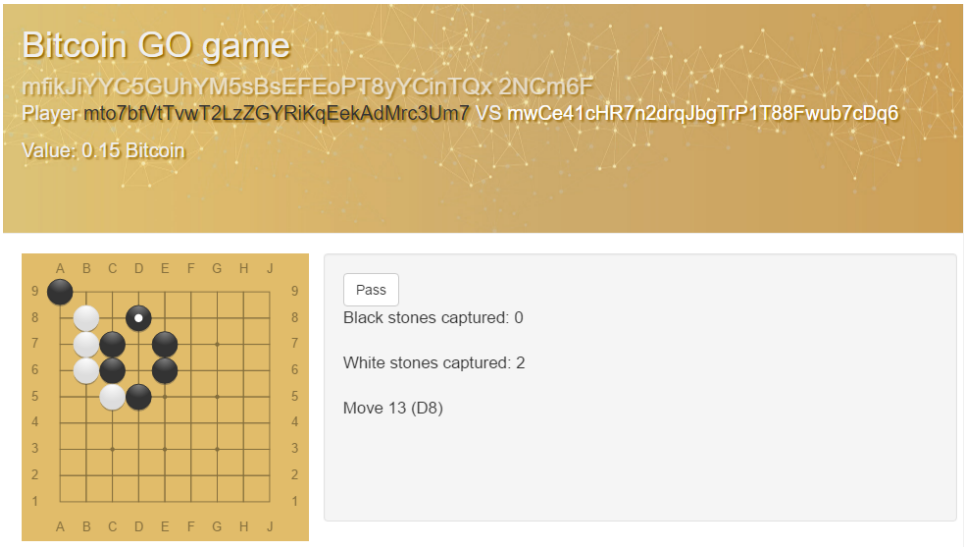


FIGURE 3.6: Screenshot of a running game

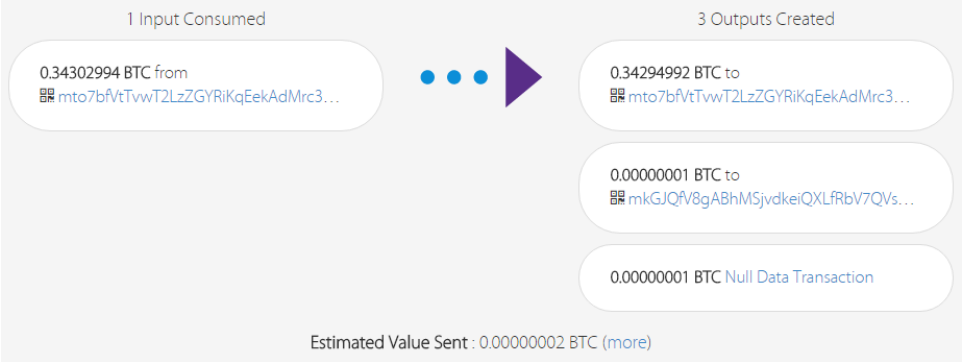


FIGURE 3.7: Sample move transaction

Source: `https://www.blockcypher.com/`

payout for the winnings in case both players do not agree on a winner and a third party – as an escrow – has to check every move. The difference of the mempool and the Blockchain will be described in more detail in chapter 4.2.3.

3.1.4 Listening to live moves

The second player listens to incoming transactions on the Blockchain under the gaming address and decodes the OP_RETURN message. It then verifies the destination address as the one of the second player and if the move is legit. When everything is correct, it places the stone on the GO board and the current turn to the second player.

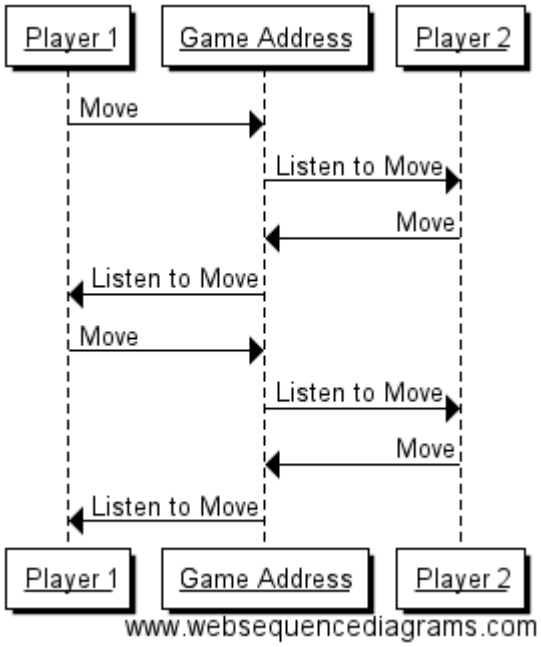


FIGURE 3.8: Sequence diagram for moves in the game

Source: `https://www.websequencediagrams.com/`

3.1.5 Replaying a game

If one player does not play live, reloads the browser, or just wants to watch an ongoing match, they must first sync up with the game from the beginning. This process is done by the backend while scanning the Blockchain and watching the fixed *mgogame...* and the random game address. It is then able to find the commands for a new game, the joining player and all moves, as well as being able to reset the game in the browser to the current state by replaying move by move.

3.1.6 End of game

When both players click "pass" or the field is full, the game comes to an end. Both players can verify who is the winner and sign a transaction to send the funds from the multisig gaming address back to the winner's wallet. After both players signed the transaction it gets submitted to the Bitcoin peer-to-peer network and the winnings are transmitted. Whenever one player does not do this or is attempting to cheat, they are not allowed to get their funds back because of the nature of the multisig address. At least two persons must sign each

transaction in the same way. The address is set up to be a 2of3 address with a third person that can audit such games and fill in the role of a referee in case of any conflicts.

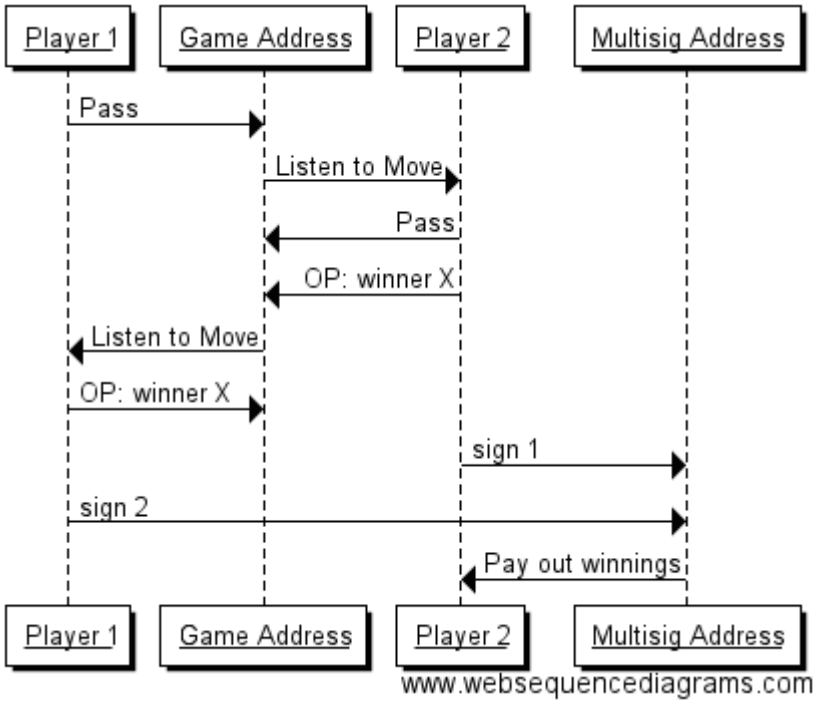


FIGURE 3.9: Sequence diagram for the games end

Source: `https://www.websequencediagrams.com/`

3.2 Legal aspects

Because the game of *GO* developed in this thesis is a form of monetary gain and arguably *gambling*, it might be important to address legal concerns.

Reading the *German Staatsvertrag zum Glückspeilwesen in Deutschland* [BD16, §3 Item 1] depicts that gambling happens when there is a random factor. That is not the case here, as *GO* is a pure strategic game lacking factors of luck or randomness. It is safe to say that this thesis has nothing to do with gambling.

Even when similar laws are not present in other countries, there is still the fact that the developed application has no middleman between two players. With the implemented Blockchain technology, there is no banker or any service controlled by a third party that

holds the users' funds. The control over the funds never leaves the user's own browser. The application does not run on any server that handles any communication for the two players, but in case any legal information is ever required, everything is publicly visible to any law enforcing agencies, thanks to the open Blockchain protocol.

Money laundering could still be a problem, as it is in many Bitcoin projects. Bitcoin's nature provides total anonymity for the user, which could be obfuscated by using a proxy server or the TOR hidden project. It is difficult, if not impossible, to track the person that bought the Bitcoins on an exchange and where they transferred those coins to.

This game could be used to transfer funds anonymously to other people. One person could just open two different browser instances and play the game by themself. However, this would just result in a transfer from wallet A to wallet B. This could also be done with a normal Bitcoin wallet without the game in between.

To be safe, the developed game only runs on the Bitcoin test network, also called Testnet [Bit16b]. This copy of the Bitcoin and Blockchain architecture is available online in the same way as the normal Bitcoin network, operating exactly the same as the main network. New features or improvements for the Bitcoin network are beeing tested for security and/or other reasons in the Testnet. This may cause it to be a problematic network to store real value in. For example, it is clearly stated that this network does not store any exchangeable monetary value. It could be resetted and restored to zero at any given time. Also, the limitation to use only the introduced standard transactions 2.3.3 are disabled here, so that this network can be used to experiment with newer or other features. Funds for testing can be obtained for free from various websites, so called *Bitcoin faucets*, given that the transactions are free too what makes the Testnet perfect for playing the game of *GO*.

Using the Testnet leaves the whole game just as a proof of concept or just as a *game* instead of legally actionable in any form.

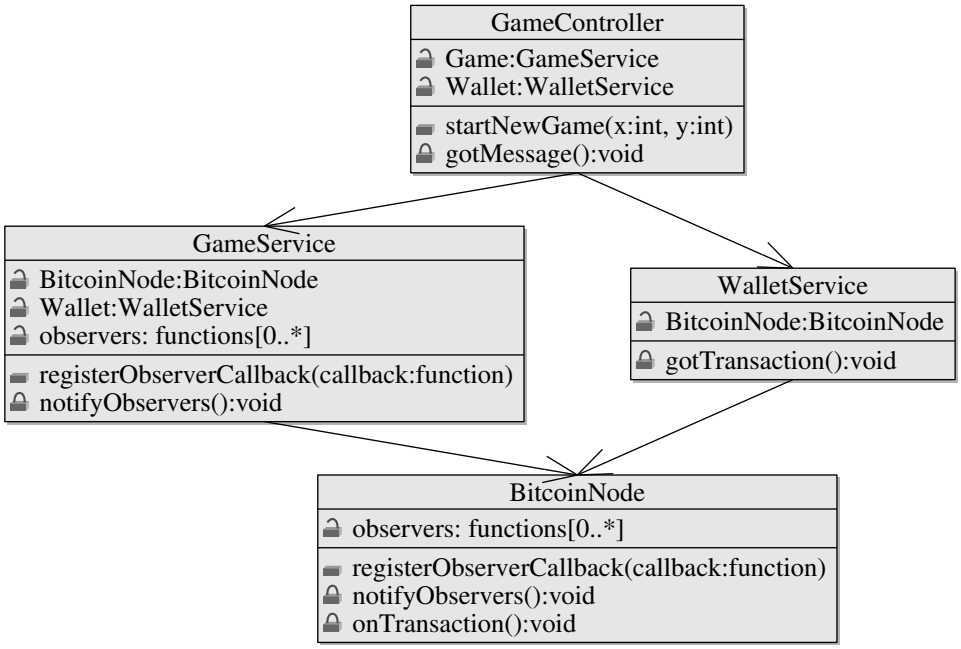
Chapter 4

Implementation

This chapter explains the technical implementation of the game and its source code. The development was by far the biggest phase in this project, because the technology is quite young and many of the used tools or practices are not well, if at all, documented. This lead to a lot of research about the basics of Bitcoin and Blockchain technology and plenty of communication with the developers of the open source tools used. The following sections detail the four biggest implemented modules and how they interact with each other. Also, the communication to Bitcoin nodes is depicted alongside the problems that occurred during the implementation phase.

4.1 Application structure

The application has four bigger modules, one serving as a controller and three as services injected to that controller. The following UML diagram shows those modules, the injections and the implemented observer patterns that are explained in the next section.



4.1.1 Angular communication between services

In the application there are four bigger modules that handle the workflow:

- GameController: This is the controller that defines the actions that can be called by the frontend and handles the communication back from the deeper services to the frontend game, like starting new games, joining, moving and passing.
- GameService: The service receives transactions from the *bitcoinNode* and utilizes the communication protocol to pass actions, such as new games and game moves to the *gameController*. Also implemented here are the smart contract, the logic to generate new games with their addresses and the player's actions like starting and joining games.
- WalletService: The *walletService* handles all balances and transactions with their UTXOs from and to the user's wallet.
- BitcoinNode: The *bitcoinNode*, as the heart of the application, handles the communication with the Bitcoin nodes and the Blockchain.

The application structure from AngularJS is based on a dependency injection pattern. Each controller or service can get other modules injected into the constructor. For this application, the *BitcoinNode* is injected in the *GameService*, and the *WalletService*, then both are injected in the *GameController*.

Because of the asynchronous communication, which can be initiated by either the frontend or the other way around from incoming transactions submitted by other players through the Blockchain, an untypical two way communication had to be implemented. This can be done in AngularJS with three big known methods:

- *$watch* on injected modules: with the usage of the *$watc*h function, AngularJS can monitor changes to attributes of function returns. This can be applied on public attributes of the injected modules. One problem with this method is that the *$watch* is only applied with a *digest cycle* and that way, can be delayed. Another issue would be intransparent communication paths.

- *$watch* on *$rootScope*: Each module can implement the so called *$rootScope* and can apply attributes to it. Other modules can consume those attributes by using the same methode previously mentioned. This is so far the most used method to handle communication between two controllers, but can lead to performance issues and unclear communication paths.

- observer pattern: With the *observer design pattern*, a function can be registered by the module that triggers the action, which can directly call this function. While this is the tidiest method to handle a communication between methods, it has some issues with the local *$scope* of the AngularJS structure. It is needed to guarantee that the "this" scope is passed to each callback functions.

For this application, I have used the observer pattern to handle the communication from the *BitcoinNode* back to the *GameController*. Whenever a transaction is sent to the service, it calls the *notifyObserver()* method. This one will call the earlier registered functions from the *GameService* and *WalletService*. The *WalletService* then checks if the transaction is sent to the user's wallet and if so, it saves the transaction ID and the incoming amount. The transaction ID is later

needed to send outgoing transactions. The *GameService* also scans the transaction, but it checks other details as well, mostly scanning for an OP_RETURN code and handling the given action in this code. Then, it calls its *notifyObservers()* function, which in turn calls the registered frontend function in the *GameController*.

4.1.2 Client communication with nodes

The first prototype was developed towards a backend API, which grants functions to listen for and send transactions with a Bitcoin wallet on the backend side. While this was quite easy to achieve and worked properly, it was not the desired goal to develop an application without the need of a trusted backend. After this first prototype it was clear that the main concepts work out and two different game instances can communicate all game moves with each other over the Bitcoin Blockchain. Then, one client was switched to a new method to interact with the full nodes directly:

The open source tool *bitcoin-net* from Matt Bell [Bel16], written for nodeJS, can be compiled with help of *browserify*[1], or *webkit*[2], to be used in any web browser. With this tool it is possible to directly connect to a Bitcoin full node and use the fundamental Bitcoin protocol.

The developed tool is acting as a *Simplified Payment Verification (SPV)* client listening only to desired transactions from full nodes with the usage of *bloom filters*. With this technology, the client specifies a filter that it listens to and sends this filter to the nodes. These nodes keep the filter into their memory and only relay transactions to the client that matches the filter(s).

The further difficulties in the communication will be described in the next chapter.

[1] `http://browserify.org/`
[2] `https://webkit.org/`

4.2 Different communication stages

There are different stages where data in the Blockchain is exchanged by the full nodes. All different stages receive the same transaction data, but need different implementations in the client that can result in delayed time-frames.

4.2.1 Live

For the live gaming between two players, their clients use the so called *inv* packets and the filtering option. Here, the communication flow is as follows: Each client sends the addresses it listens to in form of filters to its connected full nodes. After the player has made a move, the client then sends this to one of its nodes. The full node receives one transaction (game move) from client/player one and broadcasts it to all other connected nodes. Each node then applies the filters and, if the second player is also connected to it, pushes the transaction, matching the filters to its client within an *inv* packet. In this packet, only the transaction hash is included. The client then has to reply with a *getdata* packet back to the node and request the raw transaction data. Then, after it receives the data it can decode it and replay the move to the local game board.

4.2.2 Blockchain

After about 10 minutes, all transactions are packed into the next block in the Blockchain. The clients initially have to scan all new blocks in the Blockchain and replay all moves made by other clients. They do this by requesting a *headerstream* from the connected node. The nodes apply the bloom filters and reply with the desired header packets with batches of hundreds of blocks. Then, the clients request all transaction data again through *getdata* packets, as done in the live process.

The problem is that the order of the game moves are not packed into blocks in the same order as they are sent. It even could happen that one earlier move is packed into a much higher block and so is received by the clients in a different sequence or even 10, 20, or more minutes later. Resulting out of this, the transaction data has to be extended with the game move number. The game instance has

to maintain an array with all moves and only replay them when it receives the next move in the right order.

4.2.3 Mempool

When the client just jumps into the game right after the move is made and before it is packed into a block, it does not get the data in the live *inv* packets and also not during the block scanning process. There is a third way to request data from the so called *mempool*. In this pool, the nodes preserve all transactions that are not yet packed into a block by the miners. The client can request this data by sending a *mempool* packet, then receiving the same data as in the *inv* and Blockchain methods.

4.3 Description of program code

This section focuses on the implemented program code. It describes the implementations within the four modules by depicting the most important parts of it. Additionally, it describes more detailed technology parts and problems that occurred within the programming phase. The code is slightly changed for visibility reasons and some parameters (especially the testnet param) are removed to keep things simpler.

4.3.1 bitcoinNode

The *bitcoinNode* service can be considered the heart of the game. It handles all communication from the two other services to the Bitcoin network. This is implemented with the help of the *bitcoin-net* libraries from Matt Bell [Bel16]. The major problem here was that these tools are not really documentation and are in a very early development state. A lot of communication and bug-fixing had to be done to really make use of the libraries, but once they work, they provide the ability to communicate with the Bitcoin nodes in JavaScript.

One feature is that this library can act as a *Simplified Payment Verification* (SPV) client. These clients differ from full nodes in that they do not need to download and store the whole Blockchain [Bit16d]. Nakamoto writes that it is possible to only download the block headers without the whole transaction chain and relay on the next some blocks: "He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirms the network has accepted it." [Nak08]

This can be accepted as long as the nodes are not relaying false data because anyone could add a node that responds with wrong data. This could be used to break the security of a Blockchain application. For the purpose of this game, it is sufficient because both players would have to connect to the same manipulated node so that both can receive the wrong data. The chance of this happening is small enough to benefit from the reduction in bandwidth and memory. Also, the further explained *merkle tree* algorithm protects against such manipulations.

In addition to that, *SPV* clients use an interesting technique to download only specific transactions instead of whole blocks while still verifying those transactions to be valid. They use the fact that transactions are saved inside of a block with the data structure of a merkle tree. This tree saves the transaction, respectively its hashes, in the leaves and always hashes two of those hashes downwards said tree. The root node is then called merkle root (also seen in the following listing, 4.3.1). Now, when a client downloads a single transaction, it can verify the validity of that when it just downloads and compares the hashes from that leaf to the root with the neighboring hashes.

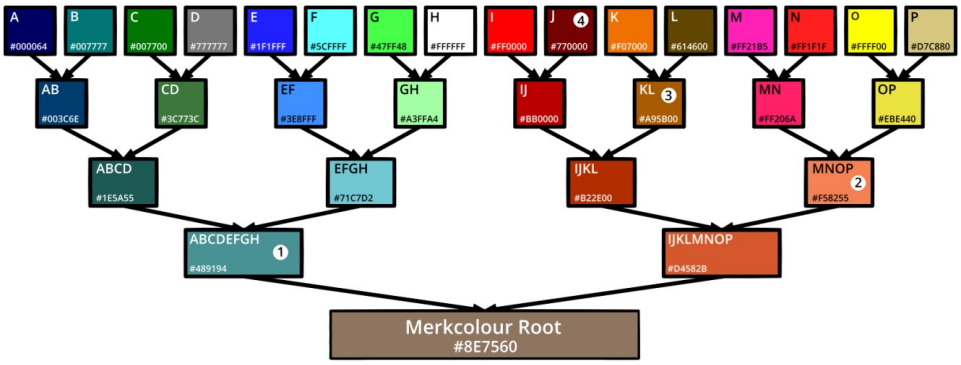


FIGURE 4.1: Merkle Tree example with colors

Source: [Har16]

For example, we take the tree with colored nodes from Fig 4.1. A client downloads the data packet I. It would then only need to download the numbered hashes 1 ABCDEFGH, 2 MNOP, 3 KL, 4 J and the merkle root to verify the validity of the packet I. First, it hashes the data in I and combine that hash with the one from J, resulting in #BB0000 as in IJ. Then, it hashes this with the downloaded hash #A95B00 from 3 KL and gets the hash in IJKL. Finally, it gets, with just some rounds of hashing, the mekle root hash without the need to download the whole data in this tree. This method makes the most sense in a distributed or peer-to-peer network where communication is more expensive than some CPU cycles used to hash the data. Merkle trees are also used for *BitTorrent* and *Git*.

SPV clients also give the advantage that they can start to download the Blockchain from any point. That reduces the initial syncing time considerably if compared to the full node, which has to download 80 GB of data for the last 8 years. In the developed application,

this is done in the *bitcoinNode* by setting a *checkpoints* variable. This is done once initially and later saved in form of the latest block to the browsers *localStorage* database. Then, it is retrieved when the user reloads the page.

```
params.blockchain.checkpoints = [
  {
    height: 927360, //latest block height
    header: {
      version: 805306368,
      prevHash: utils.toHash('000000000000009dc...06f81ce'),
      merkleRoot: utils.toHash('75ad5b2aec...ebe0dd8249'),
      timestamp: new Date('2016-09-08T10:06:07Z') / 1000,
      bits: 436339440,
      nonce: 2576579554 //nonce that this block has been mined with to match the 00000000000000 from the hash
    }
  }
];

//override the latest checkpoint in case it was saved
let latestBlock = localStorage.getItem('block');
if (latestBlock !== null) {
  params.blockchain.checkpoints = localStorage.getItem('block');
}

//receiving blocks
blockStream.on('data', (block) => {
  if (block.height % 2016 === 0) {
    //storing the block to the browsers localStorage Object
    localStorage.setItem('block', JSON.stringify(block));
  }
});
```

The rest of this service is setting up a memory database, connecting to the nodes (peers in the code) and enabling the *BlockStream* and *HeaderStreams* to start listening to blocks. Finally, it connects to the peer network and calls the registered observers once it receives a transaction.

```
//setup chain DB
const db = levelup('testChain', { db: require('memdown') });
const chain = new Blockchain(params.blockchain, db);

```

```
5  // create peer group, filter and blockStream
6  this.peers  = new PeerGroup(params.net);
7  this.filter = new Filter(this.peers);
8  const blockStream = new Download.BlockStream(this.peers);
9
10 //once connected to the first peer
11 this.peers.once('peer', () => {
12   chain.getBlockAtHeight(params.blockchain.checkpoints[0].
       height, function (err, startBlock) {
13     const readStream = chain.createReadStream({ from:
       startBlock.header.getHash(), inclusive: false });
14     readStream.pipe(blockStream);
15   });
16
17   //start HeaderStream to download block headers
18   const headers = new Download.HeaderStream(this.peers);
19   pump(
20     chain.createLocatorStream(),
21     headers,
22     chain.createWriteStream()
23   )
24 });
25
26 // notify observers with incoming transactions
27 new Inventory(this.peers).inv.on('tx', (tx) => {
28   notifyObservers(tx);
29 });
30
31 this.peers.connect();
```

The most important part to save bandwidth, computation power and memory is the ability to add filters (see line 7 in the listing above). Filters are later added by the wallet and game services to only listen to specific transactions that match the known addresses. Those filters are sent to the connected full nodes and these in turn only reply with transactions that match those. This is implemented with the following function:

```
1 subscribe(address) {
2   const hash = bitcoinjs.address.fromBase58Check(address);
3   this.filter.add(new Buffer(hash.hash, 'hex'));
4 }
```

This is a powerful feature, but lead to a problematic mistake with the described method that gave each game its own randomized game address (remember 3.1.2). Each new game is initialized to a transaction to the master address and later has its own game address to replay all moves. Combined with the ability to bulk download block headers (see the code in 4.3.1) only matching specific filters, it gives the drawback that, once the bulk download finds some game addresses for newly started games, the previous filter does not include that new games. Further blocks (game moves) are not relayed by the nodes in this bulk download because they do not match the filters and the height of the current download state is way above the previous games. This is fixed by downloading the *blockheaders* twice, once to find all games and setting up the filters for those games, and then again to receive all moves into those games. Yet, this lead to another problem: The nodes do not notify when the Blockchain reaches its end, so another solution is to wait for a timeout and then redo the whole Blockchain download. Whenever the client is connected to a slow running node, this lets the current implementation to skip some moves. A problematic behaviour that needs to refresh the page.

The implementation to the second download is similar to the listing above, but is triggered by the *blockStream.on('data')* listener, like in the initial listing, and not listed here in detail.

4.3.2 walletService

The first thing that happens when a user visits the page is the "registration" of their wallet. Typical sign up and log in features are not used here. Instead, each visitor gets a local Bitcoin wallet with a randomly generated address. For simplicity, the first version of the game only uses a single address. Later on this should be a whole wallet with many different addresses for each deposit, but because the game identifies the player by their address, it would be an additional overhead to have different addresses for each open game. The address, with its public and private keys, is then saved to the *localStorage* object of the user's browser. Whenever the user revisits the site, this object is reloaded to the *walletService*.

The address generation, setup of the filter and registration of the callback to the *bitcoinNode* are done with the help of the *bitcoinjs-lib*

[Tho+16], which works like this:

```
1 const keyPair = bitcoinjs.ECPair.makeRandom();
2 this.wallet.address = {
3   public: keyPair.getAddress(),
4   private: keyPair.toWIF()
5 };
6
7 this.listenToAddress(this.wallet.address.public);
8 this.bitcoinNode.registerObserverCallback(this.
    gotTransaction.bind(this));
```

In line 7, the public address is being listened to. This calls a function in the *bitcoinNode* service that sends a bloom filter to all connected nodes, letting them know that this client wants to be notified for incoming transactions. Now the user can deposit some Bitcoins to that address through the GUI.

After the deposits are seen by the peer-to-peer network, the first connected node with that filter sends the transaction back to the application and the *gotTransaction()* function is called.

```
1  gotTransaction(tx) {
2    tx.outs.forEach((out) => {
3      //decode out script and get pubKey from script
4      const pubKey = bitcoinjs.address.fromOutputScript(out.
     script);
5
6      //check if key is an own wallet address
7      if (this.isOwnAddress(pubKey)) {
8        //decode in script
9        let chunksIn = bitcoinjs.script.decompile(tx.ins[0].
     script);
10       let pubKeyIn = bitcoinjs.ECPair.fromPublicKeyBuffer(
     chunksIn[1]);
11
12       let prevOutTxId = [].reverse.call(new Buffer(tx.ins
     [0].hash)).toString('hex');
13
14       //sent to self, remove from unspend array
15       if(pubKeyIn.getAddress() === pubKey) {
16         this.wallet.unspend.splice(
17           this.wallet.unspend.findIndex(x => x.tx ===
     prevOutTxId), 1
```

```
18          );
19        }
20
21        //add tx and its value to the list of unspend txs
22        this.addUnspend(tx.getId(), out.value.toNumber());
23      }
24    });
25  }
```

This function scans all outputs in the transaction that belong to the user's wallet. Then, it checks if the output is the same public address as the input (this is true for all *change transactions*) and removes them from the actual list of *unspend transactions* (UTXOs). If it is not a change transaction, it must be an input from the user's deposit and as such, it is added to a local array with all *unspend transactions*. This list is used to generate outgoing transactions with a reference to the last transaction ID and the incoming value.

Then, the most important part of the *walletService* is the function to send transactions to the network.

```
1  sendTxTo(sendTos, message) {
2    //get an unspend tx or return if none is found
3    let lastTx = this.getUnspend();
4    if(typeof lastTx !== 'object') return;
5
6    //start tx and add the last unspend tx as input
7    const tx = new bitcoinjs.TransactionBuilder();
8    tx.addInput(lastTx.tx, 0); //index 0
9
10   //setup fee and calculate amount and balance
11   const fee_amount = 8000;
12   let op_amount, amount = 0;
13   if(message!=='') {  op_amount = 1;  }
14   sendTos.forEach(function(sendTo) {
15     amount += sendTo.value;
16   });
17   const balance = lastTx.value-amount-fee_amount-op_amount
18
19   //out address must be at index 0, see mirror in tx
20   tx.addOutput(this.getLatestAddress(), balance);
21
22   //add all outputs
```

```
23    sendTos.forEach(function(sendTo) {
24      tx.addOutput(sendTo.address, sendTo.value);
25    });
26
27    //add OP_RETURN message
28    if(message!=='') {
29      const dataScript = bitcoinjs.script.nullDataOutput(
30        new Buffer(message)
31      );
32      tx.addOutput(dataScript, op_amount);
33    }
34
35    //sign transaction with the wallets private key
36    const keyPair = bitcoinjs.ECPair.fromWIF(this.getWif());
37    tx.sign(0, keyPair);
38
39    //build, transmitt and save remaining UTXO
40    const buildTX = tx.build();
41    this.bitcoinNode.sendTx(buildTX);
42    this.addUnspend(buildTX.getId(), balance);
43  }
```

This function takes an array of outgoing addresses with a value and an additional message parameter. It then reads the last unspent transaction ID and the value from the user's wallet to generate a transaction with the help of *bitcoinjs-libs TransactionBuilder* by using that input. For all outputs, it adds those and subtracts the value from the input's total value. The remaining value is added to an additional output with the latest user's wallet address. Remember, the sum of all outputs must match the input transaction. If the message is added to the function, it also gets included as a null data output. At last, the transaction is signed with the user's private key and gets transmitted to the Bitcoin network. Because the change amount is sent to a new incoming transaction in the same wallet, it also gets directly added to the local wallet storage.

4.3.3 gameService

The connection of the *gameService* to the *bitcoinNode* is similar to the *walletService*. It also registers a listener to the *bitcoinNode* that listens to incoming transactions. Besides, it initializes an array with all games and the current game instance.

Because the game service handles all OP_RETURN transactions, the code to decode the transactions is more complex than the one found in the *walletService*. Decoding of input and output transactions is similar to the *bitcoinjs-lib*, but done for all ins and outs:

```
1  gotTransaction(tx) {
2    for (let output of tx.outs) {
3      const chunks = bitcoinjs.script.decompile(output.script)
4      output.pubKey = bitcoinjs.address.fromOutputScript(
       output.script);
5    }
6
7    for (let input of tx.ins) {
8      const chunks = bitcoinjs.script.decompile(input.script);
9      input.pubKey = bitcoinjs.ECPair.fromPublicKeyBuffer(
       chunks[1]).getAddress();
10   }
11
12   ...
```

After some validation checking and error handling, the loop is similar to the one in the *walletService*, with the difference that this service scans for the OP_RETURN transaction part instead of the own wallet addresses. With this transaction it reconstructs the actual game actions:

```
1  ...
2  tx.outs.forEach((out) => {
3    let game = this.games[gameAddress] || angular.copy(this.
       gameInitState);
4
5    //check if output is an OP_RETURN
6    const chunks = bitcoinjs.script.decompile(out.script);
7    if(chunks.shift() === bitcoinjs.opcodes.OP_RETURN) {
8      const message =  chunks.toString();
9
10     if(message === this.commands.new) {
11       game.state = 'open';
```

```
12        game.address.value = tx.outs[1].value;
13        game.address.public = gameAddress;
14        game.players.one = pubKeyIn;
15      }else if(message === this.commands.pass) {
16        //check for legitimate players
17        if(pubKeyIn === game.players.one
18        || pubKeyIn === game.players.two) {
19          if (game.state !== 'pass') {
20            game.state = 'pass';
21          } else {
22            game.state = 'end';
23          }
24        }
25      }else if(message === this.commands.join) {
26        game.state = 'running';
27        game.players.two = pubKeyIn;
28        game.address.paymentFromTwo = tx.outs[2].pubKey;
29
30        //player one receives the join from player two
31        if(this.wallet.isOwnAddress(this.currentGame.players.
    one)) {
32          //use all 4 pubKeys to reconstruct multiSig address
33          const pubKeys = [];
34          pubKeys[0] = new Buffer(this.masterAddress);
35          pubKeys[1] = new Buffer(game.players.one);
36          pubKeys[2] = new Buffer(game.players.two);
37          pubKeys[3] = new Buffer(game.address.public);
38
39          // generate 3 of 4 multisig address
40          const redeemScript = bitcoinjs.script.multisigOutput
    (3, pubKeys);
41          const scriptPubKey = bitcoinjs.script.
    scriptHashOutput(bitcoinjs.crypto.hash160(redeemScript))
42          const payAddress   = bitcoinjs.address.
    fromOutputScript(scriptPubKey);
43
44          //multisig payment address matches
45          if(game.address.paymentFromTwo === payAddress) {
46            game.address.payment = payAddress;
47            //forward funds from game address
48            this.wallet.spendOpenGame(gameAddress, payAddress)
49          }
50        }
51      }else{ //received a game move
52        const data = JSON.parse(message);
53        const move = {
```

```
54          x: data.x,
55          y: data.y,
56          n: data.n,
57          p: data.p,
58          pk: pubKeyIn
59        };
60        game.moves[move.n] = move;
61        this.notifyMove(move);
62      }
63    }
64    this.games[gameAddress] = game;
65  });
```

The actions new, pass and move are basically straightforward and just translate the transaction data to the *this.games* object.

The interesting part is the join command. Whenever a client receives that command, it checks if player one is the one who sent the game funds to the game address. This moment is the first time in the game that it knows its opponent and their public key. Now it can construct the same multisig address as player two did when joining the game. Finally, it can use its own private key to spend the value it sent to the game address and send it to the multisig payment address so that the smart contract behind this game can be started with both players' payment and both players' control.

After the Blockchain sync is complete and this function has constructed all games, the user can replay any previous games on the GUI. When they clicks an active game, it is possible to watch it in realtime (with *this.notifyMove(move)*) or play their moves if their wallet address matches one of the players. If only one player is set, they can join that game.

Sending game moves or a pass is a simple call to the *walletService* with:

```
1  this.wallet.sendTxTo([{address: this.currentGame.address.
       public, value: 1}], JSON.stringify(move));
```

To join a game, the deposit to the game address is added and the second player is set to the current wallet address:

```
1  this.currentGame.players.two = wallet.getLatestAddress();
2
3  //use all 4 pubKeys to reconstruct multiSig address
4  const pubKeys = [];
5  pubKeys[0] = new Buffer(this.masterAddress);
6  pubKeys[1] = new Buffer(this.currentGame.players.one);
7  pubKeys[2] = new Buffer(this.currentGame.players.two);
8  pubKeys[3] = new Buffer(this.currentGame.address.public);
9
10 // generate 3 of 4 multisig address
11 const redeemScript = bitcoinjs.script.multisigOutput(3,
       pubKeys);
12 const scriptPubKey = bitcoinjs.script.scriptHashOutput(
       bitcoinjs.crypto.hash160(redeemScript));
13 const payAddress = bitcoinjs.address.fromOutputScript(
       scriptPubKey);
14
15 this.games[this.currentGame.address.public].address.payment
       = payAddress;
16
17 const sendTos = [
18   {address: this.currentGame.address.public, value: 1},
19   {address: this.currentGame.address.payment, value: this.
       currentGame.address.value},
20   {address: this.masterAddress, value: 10000}
21 ];
22
23 this.wallet.sendTxTo(sendTos, this.commands.join);
```

The most important part is to start a new game. This is done by creating a new address for the match and sending the desired bet amount to that address. After this, the application redirects the user to that game's page and they can send the link to other people, or just wait for someone to join them.

```
1  startNewGame(betAmount) {
2    this.currentGame = angular.copy(this.gameInitState);
3
4    //generate random game address
5    const keyPair = bitcoinjs.ECPair.makeRandom();
6    this.currentGame.address.public = keyPair.getAddress();
7    this.currentGame.address.value = betAmount*100000000;
8    this.currentGame.state = 'running';
9    this.currentGame.players.one = this.wallet.
       getLatestAddress();
10
11   //add this address to the filters
12   this.bitcoinNode.subscribe(this.currentGame.address.public
       );
13
14   //send desired bet amount to that address
15   const sendTos = [
16     {address: this.currentGame.address.public, value:
       betAmount*100000000},
17     {address: this.masterAddress, value: 10000}
18   ];
19
20   this.wallet.sendTxTo(sendTos, this.commands.new);
21
22   //save this private key to be later used for forwarding
       the value to the multisig address
23   this.wallet.saveOpenGame(txID, keyPair, this.currentGame.
       address.value);
24   this.games[this.currentGame.address.public] = this.
       currentGame;
25   //redirect the browser to the game page
26   this.$state.transitionTo('game.play', {'pubKey': this.
       currentGame.address.public});
27   return this.currentGame;
28 }
```

4.3.4 gameController

The *gameController* handles the communication between the *gameService* and the frontend. It registres an observer callback to the *gameService* in the same way as the latter and *walletService* do to the *bitcoinNode*. It initializes the *tenuki lib* [Pre16] when it receives the message with the command *start* for the game and passes the commands *pass*, *join* and *move* to it. The move part guarantees that the moves are replayed in the right order and no moves are skipped. For this, the *games.move* array in the *gameService* saves the move number and the controller replays them in the *for* loop. Moves can be received in the wrong order, especially when a user reloads the site and the engine scans the Blockchain for previous moves.

```
1  gotMessage(message) {
2    if (message.type === 'start') {
3      let player = 'black';
4      if(this.wallet.isOwnAddress(message.game.players.one)) {
5        player = 'white';
6      }
7      this.resumeGame(player);
8    } else if (message.type === 'pass') {
9      this.client.receivePass();
10   } else if (message.type === 'phase') {
11     this.client.setDeadStones(message.deadStones);
12   } else if (message.type === 'move') {
13     let moveNo = this.client.moveNumber();
14     let moves = this.Game.currentGame.moves;
15
16     //replay moves with the right order and valid players
17     for(let i=moves; (typeof moves[moveNo] !== 'undefined')
18       && (i <= (moves.length-1))
19       && (move.pk === this.Game.currentGame.players.one
20       ||  move.pk === this.Game.currentGame.players.two)
21     ;i++) {
22       let move = moves[moveNo];
23       this.client._game.playAt(move.y, move.x);
24     }
25   }
26 }
```

4.4 Smart contract

The implementation of a real working smart contract was the biggest challenge in this application as there are not many readily implemented samples of this concept.

Let's take a closer look at the transactions. Besides value, destination address and the script part that unlocks the outputs, there is also the *nLockTime* and a *sequence bit*. In any normal transaction, the lock time is zero and the sequence is set to *UINT_MAX*, but for smart contracts, these two flags can be used alongside the multisig transactions. By setting the sequence bit to zero the transaction can be submitted with new values until the *nLockTime* is reached. There is also another flag called *SIGHASH* that defines how the outputs can be changed:

- SIGHASH_ALL: "I agree to put my money in, if everyone puts their money in and the outputs are this" [Bit16c]
- SIGHASH_NONE: "I agree to put my money in, as long as everyone puts their money in, but I don't care what's done with the output" [Bit16c]
- SIGHASH_SINGLE: "I agree, as long as my output is what I want; I don't care about the others" [Bit16c]

With those options, two different approaches can be used to create smart contracts between two persons. Either a transaction is passed outside of the Blockchain to the other person in an incomplete form and the second person completes it, or two transactions are used. The double transaction model has some limitations and is later explained in the Lightning Network chapter 5.2.2.

For the implemented game I have used a multisig address that uses a 2 of 3 multisig escrows approach. First, the creator of a new game generates a gaming address and sends their deposit to that address. They fully owns this address and can withdraw funds from it at any time, which would be a cancellation of the game. Then, when the second player joins, they have all information necessary to create the multisig address. The game master key is used as a third person involved. This guarantees both players to have an escrow-like service if something goes wrong. Next, their latest wallet address and the latest wallet address from the creator of the game are used.

Those three keys are always identical between those two players and out of identical keys there can be no randomized payment address. Because of this, it adds the game address that was randomly generated by player one to the now 3 of 4 multisig address. Two of those keys are controlled by player one, one by player two and one by the creator of this application as a fallback option. After such address is created, they deposits the same amount that player one has put to the gaming address. Player one then moves the funds from this to the multisig address and the game can start.

When the game ends, both players must sign the transaction that pays the winner. Player one has to sign with their two keys and provide the raw, half signed transaction to player two. They then use their private wallet key to finalize the transaction and submit it to the network. When player one or two refuse to sign the address, the master game key can be used as a security fallback. The owner of this master key then would have to look at the game and function as a judge to pay the right player.

Because this model utilizes fixed addresses and the fallback of letting player one own two of those keys, it would be mandatory to let them sign the end transaction. This can be fixed by forcing player one to generate a transaction that sets player two as the winner with a nLockTime somewhere in the future. If there's no response during that time, player two automatically wins. If player two does not respond, then the escrow mediator can jump in and fix that transaction with player one. Another way to make the signing part more reasonable for any of the losing players would be to not let the winner get 100% of the pot. If any amount still goes back to both players, there is no intention to manipulate the game code and in contradiction to always sign the transaction even if the match is lost.

Chapter 5

Results, problems and future lookouts

In the beginning of this thesis, the ambitious goal was to implement different methods to communicate over the Blockchain. I explored theoretical technologies such as Lightning Network, Rootstock, Ethereum, smart contracts and others in the desired tools for comparison. Since most of those tools were still only abstract concepts or still under development, the hope was that they would be available for testing in the middle of 2016, but in reality another problem with scaling the Blockchain to allow more transactions was slowing down the development of such techniques. This forced the development of this thesis to focus on already existing techniques, such as the OP_RETURN opcode, something that lead to problems with communicating the smart contracts and also, made the game moves quite expensive.

5.1 Problems

5.1.1 Endgame smart contract

The endgame is not yet implemented into the game because one problem is the part where player one has to pass the half signed transaction to player two. This is required by the multisig protocol and thus, an issue, because the game should not release the identity of both players to each other in form of the IP address or similar due to security and anonymity reasons. (One player could DDoS the other if they have their IP address, and so prevent them from doing further moves) First I thought to put this half signed transaction also to the Blockchain OP_RETURN data, but then I noticed that an

unsigned transaction is 800 bytes large, while in one OP_RETURN, only 83 bytes are available.

I figured out, that there can be various possibilities to solve that issue:

- Give out the IP and use *peerJS*: With that library, two browser instances can communicate with each other using a WebRTC connection, but this would break the anonymity of both players.

- Retain anonymity and split the transaction to many smaller chunks in the OP_RETURN move transactions: This would need about ten transactions and ten times the fee to be paid. It could either happen in the end of a game or the *gameService* could create two different transactions, each defining one player as the winner. Then, it could fill the open bytes in the OP_RETURN when both players do their moves with chunks of that transactions, but not the last piece. Finally, at the end only the winning player would pass the correct remaining piece to the other and let them sign it. This is a promising approach, but would need at least 20 moves between both players.

- Use another technology to provide data to a second network, like the *interplanetary file system* (IPFS)[1]: This network is built similarly to the Blockchain node network, but focuses more on saving and relaying bigger data between the nodes. It also uses the merkle tree technology and saves files in form of hashes to the network. Other users can request those files with the hash as a name from every other node. A rooting protocol finds the owner, or all nodes that already have those files, and the node relays the data. IPFS can be seen as a combination of a webserver and a merkle tree peer-to-peer network (like *Git* or *BitTorrent*, for example). With this approach, both players can put the half signed transaction in their file system and provide the required hash after the game is over. This would guarantee the anonymity of the players to each other whilst allowing less than 20 moves, but it would introduce another big JavaScript library to the already large game and the need to open an IP port

[1] https://ipfs.io/

to other nodes. According to the IPFS team, there will be an implementation of IPFS directly to one big browser engine in the near future. That would make this approach very promising.

- Using Lightning Network. Because this network does not exist yet, this is explained in the final, future lookouts chapter of this thesis in 5.2.1. This would maintain the KISS principle: "Keep it simple, stupid".

5.1.2 Open source tools bitcoin-net and bitcoinjs-lib

With, the implementation some problems arose. The chosen tool, *bitcoin-net* [Bel16], caused a lot of issues with the current state of development. It is one of the only tools that enable browsers to connect to Bitcoin nodes, but present in a very early development state. There is no available documentation and a lot of things have to be picked out of the source code. The communication with Matt Bell also took a long round trip time because of the distribution of geographical locations.

Also, the *bitcoinjs-lib* [Tho+16] left out some major documentation parts. There are some well explained examples on how to create and transmit any raw transaction, but not a single one on how to decode that back to their addresses and values. Additionally, bitcoin-net and bitcoinjs-lib use different numbered formats for the representation of the value fields, and I am certain not many people have heard of the integer formats *BN* and *INT53*.

Overall, using these tools consumed a lot of time, but hopefully the implementation in the source code present in this thesis can, in the future, help other people develop faster using the same. The source code is public available in my GitHub repository under `https://github.com/Head/gotoshi`.

5.1.3 Double scanning

One major problem with the developed protocol is the unique gaming address for each running game. While this type of separation protects the main gaming address (*mgogame...*) from being flooded with every move that is done by any player, it lead to a problem

whilst scanning the Blockchain. The clients, which listen to the *headerstream* from the nodes, receive a lot of blocks at the same time for the given bloom filters, but because they initially scan only the gaming address, they "see" all existing games, yet not all moves inside those games. To request the moves for a desired game in the past, another scan with the added bloom filter for the gaming address had to be made. In the implementation, the solution is to scan the Blockchain twice, once to find all games, then to add those to bloom filters and scan a second time to receive all moves and other actions made within those games. This would be much easier with a single centralized instance that holds all moves and can be requested for any given game, but that would lead to a loss in privacy and would make the decentralization of the implemented game obsolete.

5.1.4 Back to an API solution?

Another way to access the Blockchain data would be to switch back to the usage of an API. There are plenty of so called *Block explorers* which function as a Blockchain data access engine. Those explorers save the Blockchain as a standard (relational) database and can be accessed over a website or through an API to query blocks, transactions, or even single addresses. For instance, it would be possible to query the master game address to get a list of all transactions that represent the *start*, *join* and *end* game states. Then, when the user clicks one of the games, a new query gets executed to request all moves under that specific game address. This would provide the advantage of no necessity for the Blockchain to be double scanned. The Block explorer saves and returns all bygone transactions with that request.

The logic of how the games are stored in the Blockchain would remain the same, as well as the process of how the two players insert their moves or manage the smart contract. For the real time playing, many APIs also provide a method to hook a *websocket* to one address that gets the browser instantly notified when a new transaction appears. From the outside this would react identically to the implemented version, which takes the same information directly from the full nodes.

One drawback of this would be the delay when querying the API. Clicking one game would require the API to be requested before it could display the previous game data. Whenever that API has a downtime, it would result in a total outage of the whole game website. Lind also describes this problem in a similar fashion by saying: "As such, we found that the API could sometimes fail, refusing to accept requests for temporary periods of time. [...] In other scenarios however, we opted to forward our request to a backup API [...]. When a transaction is broadcast to the network it should ultimately be relayed to every node in the network. [...] This highlights the distributed nature of the Bitcoin network." [Lin15, p. 97] This delineates the strong advantage of the implemented method of directly communicating with the Bitcoin full node peer-to-peer network. There was not a single downtime to such network, as the existence of Bitcoin and direct connection to random nodes, the application has a 100% uptime, something that no server, API or single centralized structure can achieve.

5.2 Future lookout

5.2.1 Lightning network

The Lightning Network [PD16] is a conceptual protocol that is built on top of the Bitcoin Blockchain, currently developed by different parties [2] [3] [4]. However, to this day it is no more than a prototype promising to solve many of Bitcoin's current drawbacks:

- Confirmation time: Because the Bitcoin blocks are mined with a time-frame of ten minutes (see the chapter about mining 2.3.4), transactions are only safe to be trusted after at least ten minutes. It is even safer after two or more blocks. Currently, most applications wait for at least one or two blocks, some even six. Because it is not guaranteed that a transaction is included into the next block, this can result in waiting times of more than an hour. For example, a shop that sends out goods to customers can produce the parcel once it receives the transaction, then

[2] Lightning Network: https://github.com/ElementsProject/lightning
[3] Lightning Network Daemon: https://github.com/lightningnetwork/lnd
[4] Thunder Network: https://github.com/blockchain/thunder

check it again after one hour before it is sent out to mail. For any online business, the rights to access the application or service can be revoked after one hour if the application notices a double spent or any issue with that transaction, but for a digital download or any other form of irreversible transfer, this just requires that waiting time.

Transactions in the Lightning Network should be transmitted and trusted instantly and atomically. This could enable point of sale terminals or anything where transferences need real time confirmations.

- Scalability: Blocks in the Bitcoin protocol are hard coded to a maximum size of 1MB. Changing this would require a hard fork of the whole network, a financial risk that is not yet justified. Also, many undeveloped countries like China have a big interest in small blocks because of slow Internet connections. Bigger blocks would result in longer download times after each new block is found anywhere in the world and the next block relays on top of that. The additional time is a possible financial loss to miners. Many mining servers today are located in China.

  At the current rate, with one 1MB big block each ten minutes and a typical transaction size of 226 bytes [21.16b], this results in:

$$\frac{1\ MB}{226\ bytes} = 4425\ transactions\ per\ block \tag{5.1}$$

$$\frac{4425\ transactions}{10\ minutes} = \frac{7.36\ transactions}{s} \tag{5.2}$$

  Just seven transactions per second are alarmingly low for a global financial network, and today all mined blocks are full, which raises the transaction fees. The miners earn those fees and choose the transaction with the best fee-byte ratio.

  With Lightning, many transactions are taken out of the Blockchain system and added at the start or end of each session. This would transform the Bitcoin Blockchain from a currency to a settlement layer network. For example, the total value of

someone's bank account is in Bitcoins stored in the Blockchain. Once money is taken out of that system, they would send a transaction from their big account to buy a coffee. With Lightning in between this would result in a closer to reality scenario with cash. One transaction on the Blockchain would be made to the Lightning "wallet" with a smaller amount, then this wallet would be used to buy daily goods and transfer funds back and forth between different persons or machines. Only after this wallet is empty or the receiving party needs those stored back to the Blockchain, the session would be closed, resulting in a second Blockchain transaction.

- Micro Payments: Even though Bitcoin enables transactions with the value of just one Satoshi (or 0.00000001฿), the miners demand a fee to add this transaction to the blocks. Because of the previous issue with the full blocks, this fee is currently at five to ten cents.

  Lightning Network would enable *Micro Payment* ideas. For instance, removing ads on websites in real time by paying a small fee on each visit. Another example could be to pay a small amount to request the permission to read a full article. These fees could be much lower and paid faster, while at the same time still more valuable for the website owner than any other payment model.

Technical Lightning Network works in a system similar to the smart contract implemented in the GO game. Both parties create a multisig channel where they store funds, then they agree on spending funds out of this channel by both signing transactions with their private keys. The balance is then updated by both parties by using that double signed transaction, but is not yet transmitted to the Blockchain. Whenever a new transaction is made, the old one gets updated and again signed by both parties, invalidating the previous one. This concept is enlarged to many different parties in a form of a peer-to-peer mesh network: Each node makes smart contracts with others and transfer funds only between the directly connected ones. This makes use of the well known theory of six connections between friends and their friends. Everybody in the world can connect to anyone with only six people in between. For example, if Alice wishes to trade with

Bob, she uses her first, well known contact person, while he uses his next and so on. See Fig 5.1, x has a smart contract to w and w to v and z, but only the funding transactions are saved to the Blockchain. The latest development from September 18, 2016 is a working routing protocol, named flare [Pri+16], between 2500 simulated AWS nodes which were able to find a route in under 0.5 seconds for 80% of all tested routes[5].

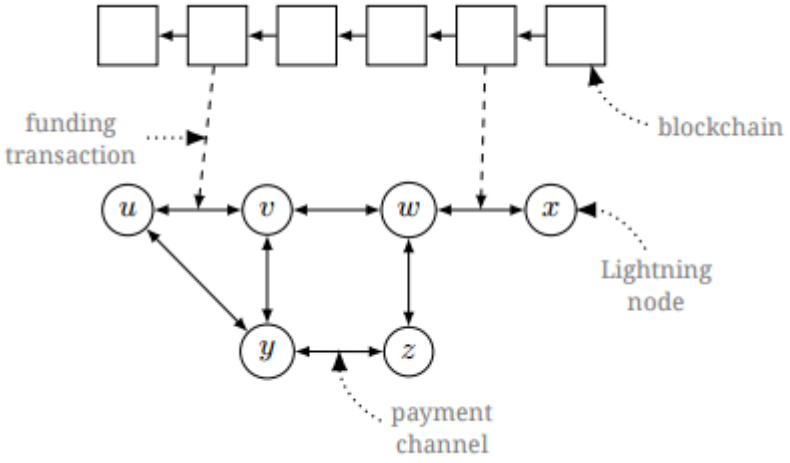


FIGURE 5.1: smart contracts in the Lighning Network

Source: [Pri+16]

Both parties together can open or close a channel at any time and transmit the signed transaction back to the Blockchain, but when one member tries to close the channel, a timeout penalty with a Blockchain feature called *nLockTime* appears. This feature saves the transaction to the Blockchain, but it is marked as unspendable until a given future time, measured in blocks. This gives the other party the time to fix that transaction in case something goes wrong. For example, Alice and Bob have an open channel with 1฿ value, both spend 0.5฿ on chain. Then, Alice transfers 0.3฿ to Bob and signs this transaction off chain (only in the Lightning Network). Bob can now transmit this transaction back to the Blockchain. There could have been thousands other transactions off chain between Alice and Bob. What happens when Alice then transmits the first transaction that claims that both parties have 0.5฿ back to the Blockchain? In this case, the

[5] https://lists.linuxfoundation.org/pipermail/lightning-dev/2016-September/000614.html

transaction has the *nTimeLock* that delays it for a validation as early as 3 days. In this time Bob can see that Alice submitted the transaction, this being able to submit all other transactions as well. Because those have a smaller *nLockTime* and both parties signed them, they get validated earlier than Alice's wrong claim, leaving her transaction to be invalidated by the Blockchain nodes. This simplified version should give an impression of how the Lightning Network will work. In detail, there are more complex algorithms with different transactions, signatures and lock times. It even has a mutual penalty system built in that makes the cheating party lose all invested Bitcoins if the other party submits all corrected signed transactions in the right order. This gives a financial incentive to both parties only transmitting the latest transactions.

Lightning Network would make things like the game in this thesis work like a charm, but it will be another half a year or more before it is usable for experiments like this. For a real world project, it will be at earliest mid till late 2017.

5.2.2 Segregated Witness

One problem why the Lightning Network is delayed that much is the implementation of a protocol called *Segregated Witness* (SegWit) [LLW15]. Primarily, SegWit was invented to reduce the space that one transaction takes in the block. To recall how transactions are stored (depicted in chapter 2.3.3), we have the signature and the public keys. The main space is used for the signatures. SegWit segregates the signature part from the transaction part and stores the signatures in a secondary tree with the same structure as the transaction tree. It then leaves the signature part in the transaction, empty. This change has a major impact on how older nodes handle these new transactions. They still process them as the TX ID is unchanged, but they cannot read the new signature part and thus, handle them as invalid. That change would need to fork the Bitcoin Blockchain.

Another part of SegWit will be that it fixes an old bug in the Bitcoin protocol that currently forces the wait for at least one transaction. This fix also enables the function to trust untransmitted chains:

"Two parties, Alice and Bob, may agree to send a certain

> amount of Bitcoins to a 2-of-2 multisig output (the "funding transaction"). Without signing the funding transaction, they may create another transaction, time-locked in the future, spending the 2-of-2 multisig output to third account(s) (the "spending transaction"). Alice and Bob will sign the spending transaction and exchange the signatures. After examining the signatures, they will sign and commit the funding transaction to the Blockchain. Without further action, the spending transaction will be confirmed after the lock-time and release the funding according to the original contract. It also retains the flexibility of revoking the original contract before the lock-time by another spending transaction with shorter lock-time, but only with mutual-agreement of both parties." [LLW15]

This is a core feature of the new Lightning Network and one reason why it is delayed that much.

Currently, SegWit is implemented in the latest version of the Bitcoin core node software with the number v0.13.0, but not yet enabled. Once a new version comes out, many users will update their node relatively fast, while older versions extinguish even faster in case of a critical update. That process can be seen in Fig 5.2.

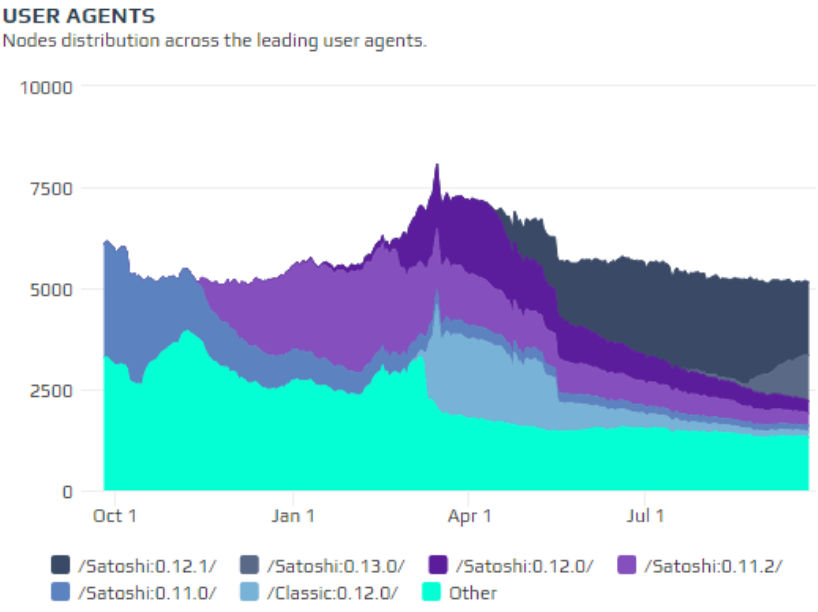


FIGURE 5.2: smart contracts in the Lighning Network

Source: [21.16a]

Chapter 6

Conclusion

6.1 Conclusion

To recall the questions from chapter 1 we have seen many examples how to use the Blockchain technology and the smart contracts in practical examples. Some are already existing, like the Namecoin domain system and some are just concepts like DRM, rental of physical goods or charging cars with machine to machine communications.

Also this game, as a proof of concept, was able to become implemented in the given time period. It is playable without leaking the players identity and is so far, the first implementation of a peer-to-peer game without any server or direct connection that I am aware of.

Looking at the questions from a developing point of view it was very difficult to program against a "database" that has this many stages for receiving data. More or less, the clients have to maintain a complete copy of the data until it decides that it has all the relevant information to continue processing. But, on the other side it makes both processes, the live and the delayed one, possible with the same backend technology. In most applications this is done with different technologies such as *socket connections* for live data, *message queues* for a temporary storage, like in the *mempool*, and databases for the final storage, like in the Blockchain part.

6.2 Final thoughts

Overall, the whole Bitcoin and Blockchain technologies are still very young. Many concepts are just ideas not yet forged into software, but the community behind Bitcoin is strong and the development is increasingly improving. It enables concepts that were yet not possible, like real distributed smart contracts, or the DAO from chapter 2.4.6.

For further research, waiting for the Lightning Network and re-evaluating makes sense if the product is in need of communicating a lot between two or many persons. The fees will be much smaller than when using OP_RETURN and the overall communication would improve a lot.

Smart contracts based on the Blockchain technology are a very powerful tool, but only make real sense when the financial instrument behind cryptocurrencies is also used and all needed data is saved and accessible on-chain. That may be with Bitcoin, Ethereum, Litecoin itself, or with any other altcoin that is not influenced by market volatility that much. For instance, this could be a special coin with a closed Blockchain. Its coins value could represent the user's Fiat money and is secured by law or legislation. The needed on-chain data could be added to the chain by extending its source code or added from any trustworthy, private agent.

The implementation of the game was a really challenging project that showed me how much is possible and how young the technology still is. Given the fact that some things are not yet possible, or just way to complicated, leaves this game just as a sample implementation and not really usable by the public. Maybe with Lightning Network some parts can still be reused to make the game worthwhile enough to be readily developed and presented to a bigger audience.

Overall, the world of Bitcoin is fascinating enough on its own, but Blockchain and Lightning Network add a new layer of fascination to me. This technology will open possibilities that were not available in both worlds of information technology and finance. An issue may be, that many people will struggle with understanding those new concepts. This can be, in my opinion, one of the core problems that hinders this technology to become mainstream. It is simply too complicated to understand how things work, but also how to implement

them to justify its use in a mass scale.

Finally, a project like IPFS can make major parties benefit from cost saving of bandwidth in the future, and so can revolutionizing the Internet in its current form of servers and cloud services. Bitcoin could do the same to money, payment systems and most of the financial world. With Blockchain and Lightning Network as a new form of decentralized, permanent data storage and a distributed, anonymous peer-to-peer protocol, not only for "the coin", but also for many other types of applications and as the basis for smart contracts.

Bibliography

[21.16a] 21.co. *Bitcoin node charts*. 2016. URL: https://bitnodes.21.co/dashboard/?days=365 (visited on 09/22/2016).

[21.16b] 21.co. *PREDICTING BITCOIN FEES FOR TRANSACTIONS*. 2016. URL: https://bitcoinfees.21.co/ (visited on 09/20/2016).

[Ant14] A.M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, 2014. ISBN: 9781491921982. URL: https://books.google.de/books?id=k3qrBQAAQBAJ.

[BD16] juris GmbH Juristisches Informationssystem für die Bundesrepublik Deutschland. *Staatsvertrag zum Glücksspielwesen in Deutschland*. 2016. URL: http://www.fst-ev.org/fileadmin/pdf/gesetze/Gesetz_2008-01-03_Gl%C3%BCcksspielstaatsvertrag.pdf (visited on 09/20/2016).

[Bel16] Matt Bell. *bitcoin-net*. 2016. URL: https://github.com/mappum/bitcoin-net (visited on 09/20/2016).

[Bit16a] BitcoinWiki. *Technical background of version 1 Bitcoin addresses*. 2016. URL: https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses (visited on 09/20/2016).

[Bit16b] BitcoinWiki. *Testnet*. 2016. URL: https://en.bitcoin.it/wiki/Testnet (visited on 09/20/2016).

[Bit16c] BitcoinWiki. *Testnet*. 2016. URL: https://en.bitcoin.it/wiki/Contract (visited on 09/20/2016).

[Bit16d] BitcoinWiki. *Thin Client Security*. 2016. URL: https://en.bitcoin.it/wiki/Thin_Client_Security (visited on 09/20/2016).

[Bit16e] Bitmaintech. *Bitmaintech ASIC miner*. 2016. URL: https://www.bitmaintech.com/product.htm (visited on 09/20/2016).

[Coi16] CoinMarketCap. *Crypto-Currency Market Capitalizations*. 2016. URL: https://coinmarketcap.com/ (visited on 09/20/2016).

[Har16] John Hardy. *What on earth is a Merkle tree? Part 2: I get more technical, but hopefully all becomes clearer*. 2016. URL: https://seebitcoin.com/2016/09/what-on-earth-is-a-merkle-tree-part-2-i-get-more-technical-but-hopefully-all-becomes-clearer/.

[Hub88] Bernardo A. Huberman. "The Agoric Papers". In: *The Ecology of Computation*. Ed. by editor. Studies in computer science and artificial intelligence. North-Holland: University of Michigan, 1988, p. 342. ISBN: 0444703756, 9780444703750. URL: http://e-drexler.com/d/09/00/AgoricsPapers/agoricpapers.html (visited on 09/20/2016).

[Laa+16] Wladimir J. van der Laan et al. *Bitcoin Core integration*. 2016. URL: https://github.com/bitcoin/bitcoin (visited on 09/20/2016).

[Lim16] DeepMind Technologies Limited. *AlphaGo*. 2016. URL: https://deepmind.com/research/alphago/.

[Lin15] Joshua David Lind. "Betting on the Bitcoin Blockchain". Master of Engineering in Computing. Imperial College London, 2015.

[LLW15] Eric Lombrozo, Johnson Lau, and Pieter Wuille. *Segregated Witness (Consensus layer)*. GitHub, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[Mol02] R.A. Mollin. *RSA and Public-Key Cryptography*. Discrete Mathematics and Its Applications. CRC Press, 2002. ISBN: 9781420035247. URL: https://books.google.de/books?id=owrOBQAAQBAJ.

[Nak08] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 09/20/2016).

[PD16] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. White Paper. institution, 2016. URL: https://lightning.network/lightning-network-paper.pdf (visited on 09/20/2016).

[Pre16] Adam Prescott. *tenuki.js*. 2016. URL: https://github.com/aprescott/tenuki.js (visited on 09/20/2016).

[Pri+16] Pavel Prihodko et al. *Flare: An Approach to Routingin Lightning Network*. White Paper. institution, 2016. URL: http://bitfury.com/content/5-white-papers-research/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf.

[Sob+08] T. Sobh et al. *Novel Algorithms and Techniques in Telecommunications, Automation and Industrial Electronics*. Springer Netherlands, 2008. ISBN: 9781402087370. URL: https://books.google.de/books?id=z8nmMkUFqdwC.

[Tho+16] Stefan Thomas et al. *BitcoinJS*. 2016. URL: https://bitcoinjs.org/.

[Tro16] John Tromp. *Number of legal Go positions*. 2016. URL: http://tromp.github.io/go/legal.html.

[Wal16] Arran Walker. *Bitcoin private key database*. 2016. URL: http://directory.io/ (visited on 09/20/2016).

[Woo14] Dr. Gavin Wood. *Ethereum: A decentralised generalised transaction ledger*. 2014. URL: http://gavwood.com/Paper.pdf (visited on 09/20/2016).

[Wui12] Pieter Wuille. *Hierarchical Deterministic Wallets*. 2012. URL: https://en.bitcoin.it/wiki/BIP_0032 (visited on 09/20/2016).