BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DISCRETE MODELING OF TRANSACTION PROPAGATION IN BITCOIN

DISKRÉTNÍ MODELOVÁNÍ ŠÍŘENÍ TRANSAKCÍ V SÍTI BITCOIN

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ MAREK**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN ZAVŘEL**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156737

Institut:	Department of Information Systems (DIFS)
Student:	**Marek Tomáš**
Programme:	Information Technology
Title:	**Discrete modeling of transaction propagation in Bitcoin**
Category:	Modelling and Simulation
Academic year:	2023/24

Assignment:

1. Study blockchain technology, focus on aspects specific to Bitcoin.
2. Study how the Bitcoin network works. Define and analyse the algorithm used to propagate created or received transactions. Consider the latest available version of the Bitcoin reference client.
3. Become familiar with discrete event simulator OMNeT++.
4. Create a design for a highly simplified Bitcoin client that can create, receive, and send messages representing Bitcoin transactions and that operates according to the algorithm from point 2.
5. As recommended by the supervisor, create a simulation model in OMNeT++.
6. Test the created simulation model and discuss the obtained results.

Literature:

1. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. 2008, 1-9 [cit. 2023-10-23]. Available from: http://bitcoin.org/bitcoin.pdf
2. Main Page. *Bitcoin Wiki* [online]. 2010, 2021 [cit. 2023-10-23]. Available from: https://en.bitcoin.it/wiki/Main_Page
3. Bitcoin Core Repository. *Github* [online]. 2009, 2023 [cit. 2023-10-23]. Available from: https://github.com/bitcoin/bitcoin
4. Learn Bitcoin and start building Bitcoin-based applications. *Bitcoindeveloper* [online]. 2009, 2020 [cit. 2023-10-23]. Available from: https://developer.bitcoin.org/index.html

Requirements for the semestral defence:
Points 1, 2 and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Zavřel Jan, Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

Blockchain technology is the core of how Bitcoin works. The aim of the theoretical part of this thesis is to describe the principles on which blockchain technology is based and also to gather information about how transactions are propagated through the Bitcoin network. The aim of the practical part is to create a model of a very simplified Bitcoin Core client that allows to create, send and receive transactions based on the propagation algorithm used in the Bitcoin network. The model is then run in the simulation environment and the simulation results are analyzed to identify the possible source node of the transaction.

Abstrakt

Technologie blockchain je stěžejním bodem fungování Bitcoinu. Cílem teoretické části této práce je popsat principy, na kterých je technologie blockchain založena a zároveň shromáždit informace o tom, jak jsou transakce šířeny v síti Bitcoin. Cílem praktické části je vytvořit model velmi zjednodušeného Bitcoin Core klienta, který umožní vytvářet, posílat a přijímat transakce na základě propagačního algoritmu využívaného v síti Bitcoin. Model je následně spouštěn v simulačním prostředí a výsledky simulace jsou analyzovány s cílem určit zdrojový uzel transakce.

Keywords

blockchain, Bitcoin, Bitcoin Core project, peer-to-peer, transaction propagation, OMNeT++, simulation

Klíčová slova

blockchain, Bitcoin, Bitcoin Core projekt, peer-to-peer, šíření transakcí, OMNeT++, simulace

Reference

MAREK, Tomáš. *Discrete modeling of transaction propagation in Bitcoin.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Zavřel

Rozšířený abstrakt

Práce se zabývá vytvořením zjednodušeného Bitcoin klienta propagujícího zprávy reprezentující Bitcoin transakce, které jsou následně monitorovány. Zachycená komunikace je poté analyzována s primárním cílem zjistit zdrojový uzel transakce.

Tato práce je rozdělena do dvou hlavních částí obsahující nejdříve kapitoly zaměřené na teorii a následně v druhé části praktické kapitoly týkající se návrhu, implementace a analýzi. Cílem teoretických kapitol je poskytnout úvod do technologie blockchain a jejích hlavních součástí, jako jsou bloky, peer-to-peer sítě, mechanismy konsensu a potenciální výskyty forků. Dále se práce věnuje první a nejznámější kryptoměně, Bitcoinu, přičemž se zabývá především strukturou transakce a tím, jak jsou transakce v sítí Bitcoin šířeny. Cílem praktických kapitol je navrhnout, implementovat a analyzovat simulační model velmi zjednodušeného Bitcoin klienta, který dokáže šířit zprávy reprezentující Bitcoin transakce podle algoritmu používaného v reálné Bitcoin síti.

První teoretická kapitola se zabývá technologií blockchain, díky které funguje většina kryptoměn po celém světě. Kapitola popisuje tři základní stavební kameny blockchainu, kterými jsou blok, řetězec a peer-to-peer síť. Další probíranou a nedílnou součástí je konsensus mechanismus, který slouží v peer-to-peer sítích k zajištění toho, aby se všechny uzly v sítí shodly na stejné verzi používaných dat. V poslední řadě jsou zmíněny i různé typy blockchainů a druhy forků, které mohou v síti nastat.

Druhá teoretická kapitola je zaměřena na celosvětově známou digitální měnu Bitcoin. V této kapitole je popsáno co jsou to transakce a k čemu v Bitcoin síti slouží. Nadále je popsána jejich struktura a skripty, které určují za jakých podmínek mohou být bitcoiny v transakci utraceny. V každém případě platí, že než mohou být utraceny, musí se po síti rozšířit propagačním algoritmem, který je popsán v předposlední sekci o Bitcoinu. Šíření zpráv exponenciální funkcí popsané v této sekci je nadále využíváno v praktické části této práce. Na závěr této kapitoly je uveden princip a příklad náhodně generovaného čísla, které musí těžaři Bitcoinu uhodnout, aby vytvořili nový blok v blockchainu.

Součástí práce je rovněž stručný popis simulátoru diskrétních událostí OMNeT++. Hlavním účelem simulátoru je modelovat a simulovat složité počítačové a komunikační systémy. V praktické částí této práce je simulátor využíván pro simulace Bitcoin sítě.

Implementační část práce spočívá v návrhu zjednodušeného modelu Bitcoin klienta, který dokáže vytvářet, přijímat a odesílat zprávy reprezentující Bitcoin transakce a následné simulaci reprezentující Bitcoin síť. Způsob propagace transakcí je implementován algoritmem pro šíření inventory zpráv z Bitcoin Core klienta, který je popsán v teoretické části. Tento klient je nejpoužívanějším Bitcoin klientem na světě a z tohoto důvodu byl také použit [12]. Na základě návrhu je model implementován v již zmíněném simulátoru OMNeT++. Součástí vytvářené Bitcoin sítě je navíc monitorovací uzel, který je připojen ke všem dostupným uzlům. Monitorovací uzel slouží k ukládání informací o příchozích inventory zprávách od ostatních uzlů, ze kterých po dokončení simulace vytvoří CSV soubor. Samotnou simulaci je možné pouštět vytvořeným automatizovaným skriptem s volitelnými parametry určující strukturu a chování sítě, což je popsáno na začátku kapitoly o implementaci.

V závěrečné částí práce popsané v poslední kapitole je cílem analyzovat zachycené informace z CSV souboru. Mezi zachycené informace patří číslo běhu simulace, transakční hash identifikující konkrétní transakci, jméno souseda, od kterého byla daná transakce obdržena a čas ve kterém byla transakce obdržena monitorovacím uzlem. Z těchto záznamů jsou následně získány nejrychlejší uzly z pohledu rychlosti propagace transakcí a jejich pořadí.

Hlavním cílem je určit potenciální zdrojový uzel transakce. Vedlejším cílem je celkově analyzovat chování sítě při různém počtu zadaných parametrů.

Discrete modeling of transaction propagation in Bitcoin

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Jan Zavřel. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Tomáš Marek
May 5, 2024

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Ing. Jan Zavřel, for his guidance, support, valuable feedback, and expertise throughout my research process. I also want to thank my girlfriend Barča, for her love and support, which kept me motivated and focused, even during the most challenging times, and also for understanding that I had to spend a lot of time sitting in front of a computer.

Contents

Chapter 1

Introduction

The first and most known cryptocurrency, known as Bitcoin, was publicly shared in 2008 by Satoshi Nakamoto. Today, there are thousands of cryptocurrencies around the world. Most of them are using blockchain technology for their functioning. In the future, we can expect blockchain technology to be used even more in other spheres and not only in cryptocurrencies.

This thesis deals with a theoretical introduction to blockchain technology, where the major parts of blockchain like block, peer-to-peer network, and consensus mechanism are described. Furthermore, consideration is given to the various types of blockchain or forks that may arise when employing blockchain technology.

The third chapter summarizes the Bitcoin cryptocurrency. The main focus here is on the transaction structure and what happens with the transaction, when a node receives it. The propagation algorithm and the exponential function used in Bitcoin Core clients to propagate transactions are also integral parts that are described. Moreover, the most commonly used scripts in Bitcoin are described together with the addresses whose format is based on the script that is being used. At the end of the Bitcoin section, the presence and calculation of the random number that miners use in the Bitcoin mining process is described. In addition to the theoretical explanation, some of the principles are demonstrated with concrete examples to better understand the described technology or process.

The practical part describes the implementation of a highly simplified Bitcoin client model. The model allows to create, receive, and send messages representing Bitcoin transactions based on the propagation algorithm. The model is created in the OMNeT++ simulator, described briefly in the fourth chapter.

The goal of this thesis is to explain and summarize the basic concepts of blockchain technology and how it works. The second goal is to describe the Bitcoin cryptocurrency, its transaction structure, and the process of propagating received or newly created transactions. The last goal is to create a simplified Bitcoin client in the OMNeT++ discrete-event simulator, which will propagate messages that represent Bitcoin transactions using the Bitcoin Core propagation algorithm. Finally, the transaction propagation is analyzed to find the fastest nodes in the Bitcoin network and the possible source node of the transaction.

1.1 Structure

- Chapter 2 describes the structure of blockchain technology, types, consensus mechanism, and type of forks.

- Chapter 3 contains an introduction to Bitcoin cryptocurrency. It mainly focuses on its transaction structure and the propagation of received transactions. Furthermore, a description of the scripts and the meaning of Nonce is described.
- Chapter 4 provides a basic overview of the OMNeT++ simulator.
- Chapter 5 describes the design of the Bitcoin simulation network.
- Chapter 6 presents the implementation of the Bitcoin simulation network, the scripts related to its generation, analysis and the verification and validation of the created simulation model.
- Chapter 7 reveals the results obtained from the simulation analysis.
- Chapter 8 concludes the thesis with a summary of the most important objectives and their achievement. The contribution of the developed simulation model and its possible future improvements are described.

Chapter 2

Blockchain

Blockchain is a digital ledger that is often publicly accessible to all participants in the network. In most cases, it is a decentralized system that acts as a distributed database, which means that there is no central authority that controls the system. The distributed nature of the blockchain ensures that the network remains operational even if individual components fail.

The blockchain operates on a peer-to-peer network. A peer-to-peer network is mainly a decentralized system where participants can share resources directly with each other without the need for a centralized authority. The fundamental components of the blockchain are blocks. Each block is connected to the previous one, creating a chain of blocks. Three main elements, Block, Chain, and Peer-to-Peer network, used in the blockchain are described in Sections 2.1, 2.2, 2.3, and are based on Bitcoin Wiki [2] and Bitcoindeveloper [11, 7]. As a side resource for understanding blockchain technology, the Hyperledger Foundation White Papers [17] were used.

2.1 Block

In blockchain, a block is a unit of data that represents a collection of transactions, where each block is cryptographically linked to the previous block. The blocks are verified and added to the chain through a consensus mechanism, which is described in Section 2.4. Once a block is added to the chain, it cannot be altered. The only way blocks can be deleted is when there are more branches in use. This situation can occur when an accidental fork happens. This process is described in Section 2.6. The very first block in the blockchain is named the Genesis block. This block is not added by users, but is standardized in the blockchain protocol.

When creating a block, several pieces of information are included. The specifics may vary depending on the type of block you are working with, but the general structure of a block is the following:

- **Blocksize** - Sets the size limit on the block.
- **Block header** - Contains information about the block.
- **Transaction counter** - Number representing how many transactions are stored in the block.
- **Transactions** - a list of all transactions within a block.

The block header contains sub-elements which are:

- **Version** - The cryptocurrency version that is being used.
- **Previous block hash** - Contains a hash of the previous block header.
- **Hash Merkle root** - The final hash is known as the Merkle root of a Merkle tree in the current block. A Merkle tree is a data structure that is used to efficiently store and verify large amounts of data in a blockchain. It works by hashing pairs of data repeatedly until a single hash remains, known as the Merkle root. The structure of the Merkle tree hashing is shown in Figure 2.1.
- **Timestamp** - The approximate time when the block was created or mined. The format can be different for some cryptocurrencies. For example, Bitcoin uses the timestamp in Unix format, which represents the number of seconds that have elapsed since January 1, 1970, at 00:00:00 UTC.
- **Bits** - The difficulty rating of the target hash, signifying the difficulty in solving the nonce.
- **Nonce** - Nonce is a 32-bit number and a shortcut for a *Number used only once*. It is part of the consensus mechanism that miners use to find valid blocks and earn rewards. A nonce is a random number that miners add to the block data and then hash it. The resulting hash must meet specific criteria.

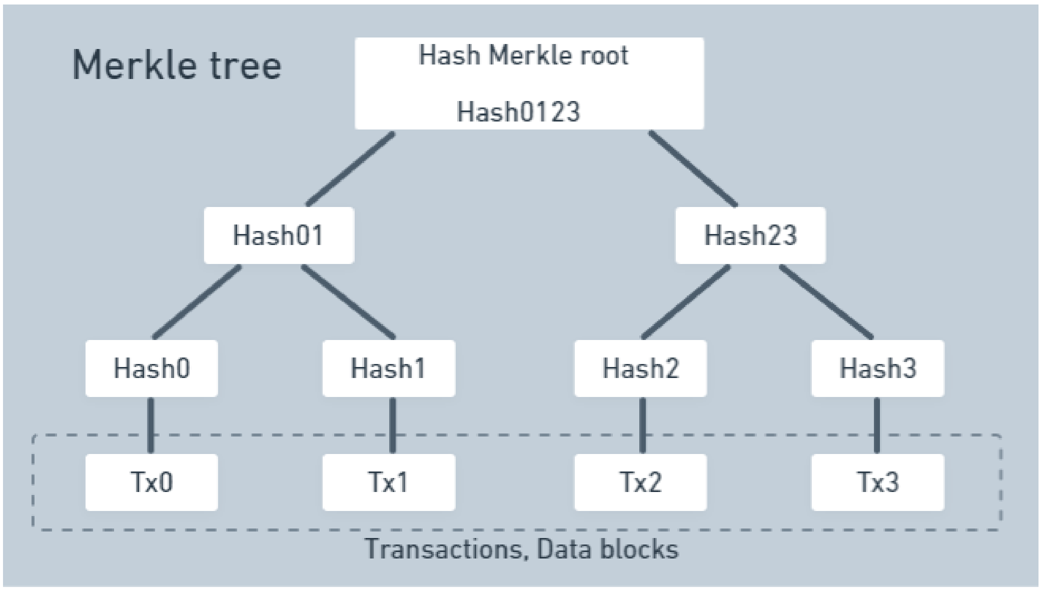


Figure 2.1: Merkle tree system hashing each pair of data until one single hash remains, known as the Merkle root.

2.2 Chain

A chain is a linked sequence of blocks. Each block contains a cryptographic hash of the previous block header, creating a chain of blocks. This chain is what allows the blockchain to function and create trust through mathematics. The hash is generated from the data that was present in the previous block, acting as a fingerprint and locking the blocks in order and time. There are various hashing algorithms used in blockchain technology. One of them is SHA-256, where SHA stands for *Secure Hash Algorithm* and 256 means fixed-size 256-bit hash. This algorithm is used, for example, in Bitcoin. On the contrary, the Ethereum cryptocurrency uses the Keccak-256 algorithm, which is part of the SHA-3 family [19].

2.3 P2P Network

Peer-to-peer network, also known as P2P, is a type of network architecture model. If a group of participants share some of their hardware resources such as processing power, storage, network link capacity, etc., a distributed network architecture can be described as a peer-to-peer network. Members of a P2P network are called nodes or peers. A node represents a single computer or device linked with other systems over the Internet. All nodes are considered to be equal in the peer-to-peer network. P2P network has different usages compared to the client-server model. The nodes share resources and information directly without the need for a central server. Each participant acts as both a client and a server, which makes the network more decentralized and less dependent on a single point of failure. To demonstrate this, think about a situation where the peer is connected to another peer, which suddenly disconnects. In that case, the peer can request another peer for cooperation. Once the connection is established, the peers can continue where the work ended with the previous peer. This is a significant advantage over the client-server architecture, where communication is lost after the server disconnects.

There are several types of P2P networks, and some cryptocurrencies use more than one P2P network. One such example is Ethereum, which uses both structured and hybrid types. Bitcoin, on the other hand, uses only a structured P2P network. The overview of P2P network types is described below and is based on [18, 24].

2.3.1 Structured

A structured network employs a specific protocol to organize network nodes into a structured overlay network. Each node is assigned a unique identifier, which is used to organize the network nodes into a logical structure, such as a distributed hash table (DHT). This structured overlay network provides efficient routing and lookup mechanisms for data stored on the network.

In a structured P2P network, each node maintains a routing table, which contains information about other nodes in the network. The routing table is organized based on the identifier space, and each node is responsible for maintaining information about a subset of the identifier space. As a result, structured P2P networks provide efficient routing and lookup mechanisms, even in large-scale networks. An example of a structured P2P network is Ethereum's consensus layer or Bitcoin.

2.3.2 Unstructured

In an unstructured network, there is no fixed structure or organization. Peers are free to join or leave the network anytime without affecting its functionality. To communicate with other peers, they broadcast their queries or messages. They hope to find the desired file or peer, but they have no information about the location of peers. In this type of network, peers maintain a fixed number of connections with their neighbors. An example of an unstructured P2P network is the first decentralized P2P network Gnutella [1] or the very first commercial P2P network called Napster [2].

2.3.3 Hybrid

a hybrid P2P network combines structured and unstructured topologies. The peers are divided into different groups or clusters, based on their characteristics, such as location, bandwidth, or performance. Each group or cluster has a leader or a super-peer, which acts as a mini-server or an index for the group or cluster and connects to other leaders or super-peers in the network. The peers contact their leader or super-peer to search and route all their queries with other peers within or across groups or clusters.

Ethereum's execution layer is an example of a hybrid network that consists of two stacks: the discovery stack and the DevP2P stack. The discovery stack is based on UDP and helps a new node locate peers to establish connections. On the other hand, the DevP2P stack is based on TCP and allows the nodes to share information with each other. These two stacks operate concurrently and complement each other to improve the efficiency of the network [26].

2.4 Consensus Mechanism

The consensus mechanism plays a vital role in ensuring that all nodes within a network agree on the same version of a transaction or a piece of data. The consensus ensures the integrity of the network, but also helps prevent any malicious activities that could compromise the system. There are various consensus mechanisms in use today, and the most known are Proof of Work (PoW) and Proof of Stake (PoS). Ultimately, the goal of any consensus mechanism is to establish trust and provide a reliable network for all nodes. Once most of these nodes agree on the same, it is noted as truth, and consensus is reached. Figure 2.2 shows the concept of how blockchains come to an agreement after a user requests a transaction. This section itself is based on [11, 19, 21].

2.4.1 Proof of Work

The Proof of Work (POW) algorithm was initially intended as a way to combat email spam and prevent denial-of-service attacks in the Hashcash system. The basic idea behind their proposal was to require senders to perform a certain amount of computational work before sending an email [4].

The POW in the blockchain is used to validate transactions and blocks. The validation depends on the miners who compete to solve the nonce. Once the nonce is correctly guessed, a new random number is generated, which will be used in the next guessing process. This cycle repeats itself as miners continue to compete in solving the encrypted numbers to get rewards.

The process of solving the number is called *block mining*. This process requires a substantial amount of computing power and consumes a significant amount of electricity resources. Anyone on the P2P network can actively participate in this block mining process. Participants can be individual computers, but the rise in mining difficulty led companies

[1] https://gnutella3.sourceforge.net/
[2] https://www.napster.com/us/

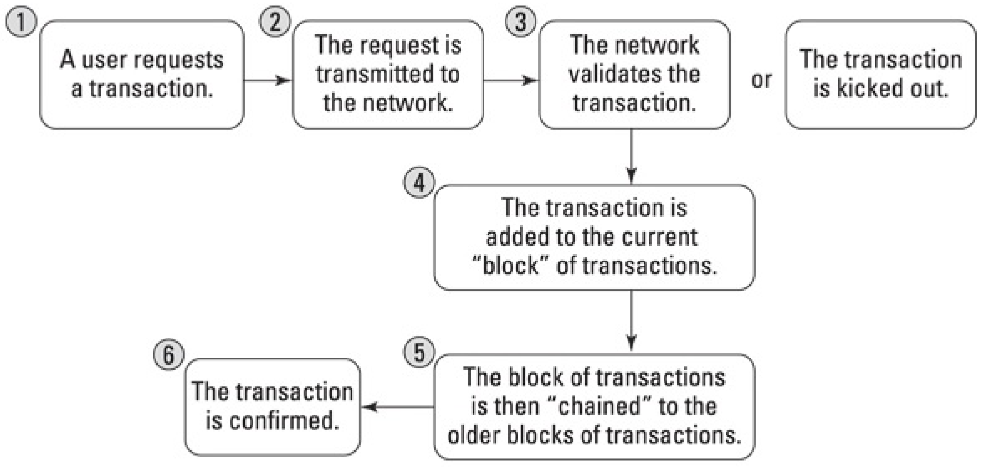

1
A user requests a transaction.
2
The request is transmitted to the network.
3
The network validates the transaction.
or
The transaction is kicked out.
4
The transaction is added to the current "block" of transactions.
5
The block of transactions is then "chained" to the older blocks of transactions.
6
The transaction is confirmed.

Figure 2.2: Process of requesting and verifying a transaction (taken from Blockchain For Dummies [19]).

to develope specialized computers called Application-Specific Integrated Circuit (ASIC) machines. ASICs are designed specifically to perform POW computations.

Some of the cryptocurrencies that use POW as a consensus mechanism are Bitcoin, Litecoin, Dogecoin, and many more.

2.4.2 Proof of Stake

Proof of stake (POS) is an alternative consensus method to verify transactions and add new blocks to a blockchain. This method is considered to be more energy-efficient than POW since it does not need that much computing power. The validation of new blocks depends on the number of coins that are staked. This process is also known as *block minting.* The word "minting" implies that you need to have a currency reserve as a backup to mint your coins. This is similar to how some central banks keep a gold reserve to support their national currency printing. With PoS, the minters stake their ownership in the system as a security deposit.

The selection algorithm chooses the validators. Once the validator is selected, they have the exclusive right to create a block. The other validators that are not currently selected to create a block are essentially in standby mode. They are not actively participating in the process of block creation at that time. Instead, they typically monitor the network to ensure that the selected validators are acting correctly and following the rules of the protocol. To prevent and punish such actions as breaking the rules, PoS uses a security function called „slashing". If someone breaks the rules, some of the crypto coins they staked will be destroyed.

Cryptocurrencies that use POS as their consensus mechanism are, for example, Ethereum, Solana, or Cardano.

2.5 Types of blockchain

First of all, it is important to define terms such as centralization and decentralization. The definition of centralization in the book Building Decentralized Blockchain Applications is: *„the act of consolidating authority to one central place“* [25]. For example, the architecture model client-server is based on centralization because the data are stored only on the server. The opposite is decentralization, which is based on operating without a central point of control or authority. The definition in the book says that the decentralized approach is when: *„there is no central authority involved in the working of Blockchain“* [25]. An example of decentralized technology is the P2P network.

Blockchain technology can be divided into different types. The first two types to compare based on access are public and private blockchains. In the public blockchain, users can remain anonymous and each user can have a copy of the ledger. Anyone can join the public blockchain. In contrast, the private blockchain requires users to provide credentials, and only authorized users have access to the ledger.

In terms of participation, the blockchain can be further divided into a permissioned and a permissionless blockchain. The permissioned blockchain does not require PoW to validate transactions since the institution (e.g. bank) provides the trust. Even without the PoW or PoS consensus mechanism, the permissioned blockchain still has the following functions:

- Privacy - only members have the right to view the transactions
- Scalability - can be easily scaled up by not using the resource-intensive PoW
- Access Control - the access to the data within the ledger can be restricted as the owner desires

In summary, the permissioned blockchain is a closed, private blockchain with an owner with a certain degree of centralization.

For the permissionless blockchain, the situation may be different. It is public, so anyone can see the transactions in the ledger and there is no single institution that provides the trust. Trust is gained through consensus mechanisms. An example of a permissionless blockchain is the Bitcoin or Ethereum cryptocurrencies. This section was based on [21, 25].

2.6 Forking

Blockchain forking is the process of creating two or more separate blockchain networks, each with its own set of rules and protocols. It may occur due to a variety of factors, including network upgrades, software updates, community disputes, or attacks on the blockchain network. There are various types of forks, including hard forks, soft forks, codebase forks, and accidental forks. This section is based on [11, 23].

2.6.1 Accidental forks

An accidental fork occurs when two or more miners find a block at roughly the same time. The blockchain is temporarily split, and various nodes may have different copies of the ledger. The fork is resolved when subsequent blocks are added, and one of the chains becomes longer than others. Blocks that are dropped by the network because they are not in the longest chain are called orphaned. Accidental forks typically do not last long and the impact on the network is negligible. The process is described once more in Figure 2.3.

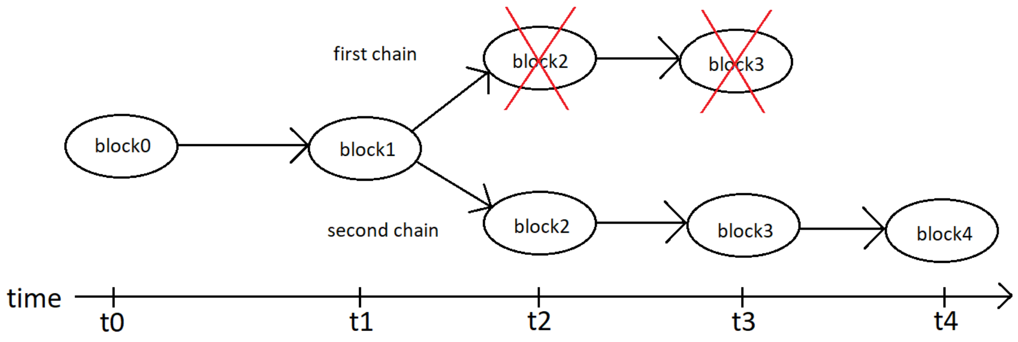


Figure 2.3: An accidental fork happened, because the **block2** was created by two miners at the nearly same time, but each with a different structure. In that case, the chain was split into two chains. Since the second chain became longer than the first one, due to **block4**, it is chosen as the winner. The **block2** and **block3** from the first chain are deleted from the blockchain database and called orphaned.

Forks can also be used for double-spending attacks. These attacks consist of trying to spend the same digital currency twice. The steps below outline how this attack, which may seem like an accidental fork, occurs.

1. An attacker makes a purchase and receives delivery of the purchased item. His transaction is now added to the blockchain.
2. The attacker creates a new longer chain, omitting his transaction.
3. If he succeeds, the attacker has the purchased item and the coins he spent on that.
4. The attacker may then spend his coins again.

2.6.2 Hard forks

a hard fork is a modification of the blockchain protocol that is not backward compatible and requires software updates from all users to continue using the network. Users who do not update their software will not be able to validate new blocks that follow the updated rules. A hard fork causes the network to split into two different versions. The original uses the old rules, and the new uses the new rules.

Example of a Hard fork

An example of a hard fork is a situation that occurred in July 2016, when the Ethereum blockchain experienced a hard fork. The fork was the result of a disagreement among the Ethereum community about how to handle a major security breach known as the DAO attack that resulted in the loss of more than 45 million dollars. The hard fork resulted in the creation of two separate blockchains: Ethereum (ETH) and Ethereum Classic (ETC). The new Ethereum blockchain continued with the updated protocol that addressed the security breach, while the original Ethereum blockchain continued with the old protocol. Those who supported the new protocol moved their ether to the new blockchain, while those who opposed the change stayed on the old blockchain, which became known as Ethereum

Classic. On the Ethereum Classic blockchain, you can clearly see the DAO attack, which occurred on block 1 757 821 [13].

2.6.3 Soft forks

a soft fork is a change in the blockchain protocol that is backward compatible and allows the introduction of new rules without requiring all users to update their software. Nodes that are not up-to-date are still able to process transactions and add new blocks, as long as they do not break the new rules. An example of a soft fork is SegWit in Bitcoin, where the transaction format was changed or the introduction of Pay-to-Script-Hash, which will be described later in this thesis.

2.6.4 Codebase forks

a codebase fork occurs when the entire source code of a blockchain project is copied and modified to produce a new piece of software or product. The original blockchain network is not affected by this, but a new one is created with new features and objectives. An example of a codebase fork is Litecoin, which was created from Bitcoin.

Chapter 3

Bitcoin

Bitcoin is a digital currency that was released to the world in October 2008 through the White paper [22] by an anonymous person or group named Satoshi Nakamoto. The genesis block of Bitcoin was mined in January 2009. From that time, Bitcoin went from the only cryptocurrency to the most known digital currency in the world. It is a decentralized permissionless currency. Bitcoin was released at the time of the 2008 financial crisis, which highlighted the need for a currency that was not dependent on third parties, such as banks.

Bitcoin was originally intended to be used as a payment system, similar to banks but without the bank itself. However, due to the value of bitcoin changing its price, it became more popular as an investment product than a daily payment system. The price of a Bitcoin is determined by the balance between supply and demand. When the demand for bitcoins increases, the price also increases, and when the demand falls, the price falls as well. Since there is only a limited number of bitcoins in circulation and new bitcoins are created at a predictable and decreasing rate, demand must follow this level of inflation to ensure price stability. The smallest piece of Bitcoin value is Satoshi, named after the founder of Bitcoin. The frequently used word for Bitcoin value is bitcoin with the shortcut BTC. One BTC is equal to 100 million Satoshis. In total, there can be up to 21 million BTC available in the future, which makes Bitcoin limited. The last Bitcoin is expected to be mined around the year 2140. New bitcoins are added to the Bitcoin supply approximately every 10 minutes, which is the average amount of time it takes to create a new block of Bitcoin. These blocks are filled with transactions that are described in the next Section 3.1.

To transfer a certain amount of BTC, a user must create a transaction. These transactions need to be transferred, verified and stored somewhere. That is the moment when Bitcoin utilizes blockchain technology. As Chapter 2 describes, the blockchain is a distributed ledger that records all transactions made on the network. All Bitcoin transactions are stored in this blockchain ledger. The consensus mechanism for Bitcoin, called Proof of Work, also relies on blockchain technology to achieve agreement among nodes on the validity of transactions. Bitcoin protocol is running on the Bitcoin P2P network, where computers are connected around the world. These computers are called Bitcoin nodes.

Bitcoin is a continuously evolving technology that undergoes regular improvement. To suggest any changes or additions to the Bitcoin protocol, the Bitcoin Improvement Proposals (BIPs) were introduced. BIPs provide a structured way of submitting new ideas or changes to the Bitcoin community for review and consideration. These proposals can range from minor technical improvements to significant changes in how Bitcoin operates. The entire process of submitting a BIP is outlined in BIP 2 [14]. Currently, there are almost 400 published BIPs as of January 2024, which are readily available on the Bitcoin GitHub repository[1]. Since Bitcoin is an open source software, the source code for the Bitcoin client [5] is also available on GitHub. Anyone can access, review, and contribute to the development of Bitcoin software. The code is constantly evolving, with regular updates and improvements made through the BIPs described above.

The source of information for this introduction to Bitcoin was based on the Bitcoindeveloper website [1] and the book Grokking Bitcoin [23].

3.1 Transactions

Transactions allow users to transfer Bitcoin values. They are broadcast to the network and collected into blocks. All communication is done over the TCP protocol. Transactions are not encrypted. This makes it possible to browse and view every transaction ever collected into a block, and everyone can verify them because they are visible.

The standard process of a transaction from the send-to-receive state involves four steps.

1. The sender initiates a Bitcoin transaction by creating a digital signature using their private key, entering the data from the scriptPubKey (mostly representing the bitcoin address) that was sent by the recipient and the amount of BTC to be sent.

2. The transaction is then broadcast to the Bitcoin network, which consists of a decentralized network of nodes that verify and process transactions.

3. Bitcoin miners compete to solve complex mathematical puzzles to add the transaction to a block on the blockchain. Once a miner solves the puzzle, the block is added to the blockchain, and the transaction is permanently recorded.

4. Once the transaction is confirmed and added to the blockchain, the recipient can access the Bitcoin sent to their address. They can then choose to hold on to Bitcoin as an investment or exchange it for another currency.

Example

Before describing the transaction structure, an example of the Bitcoin payment process is demonstrated. In this example, John wants to send 1 BTC to Thomas. Inspiration for this example was taken from Grokking Bitcoin [23]. The names John and Thomas are used for simplicity, but in the real world, Bitcoin does not use any names or personal information. The payment process can be split into 4 steps.

1. **Transactions** - The process starts when John requests the network to send 1 BTC to Thomas. That is done by sending a Bitcoin transaction to the Bitcoin network through a mobile wallet application. This transaction includes a piece of information that describes:

   - The amount of bitcoins to be transferred.
   - Thomas's Bitcoin address, where the money is sent.
   - a digital signature from John to prove that it is him who wants to send the money.

[1] https://github.com/bitcoin/bips

2. **The Bitcoin Network** - Once John has sent a transaction to the Bitcoin network, Bitcoin nodes check and verify if the transaction is valid. The verification process is done by consulting its local copy of the blockchain ledger, checking if the 1 BTC that John spends exists and if the digital signature is valid. Invalid transactions are dropped, and the valid ones are forwarded by nodes to their peers. The blockchain ledger has not been updated yet because it will be done in the next step.

3. **The Blockchain** - As there can be up to thousands of transactions waiting to be added to the blockchain, one node must take the lead and send a message to the network about which transactions he will add to the block. The other nodes verify the block and update their blockchain copies. John's transaction is part of this block and is now part of the blockchain.

4. **Wallets** - All users who want to participate need a computer program to interact with the network. This program is called a Bitcoin wallet. Since John's transaction is part of the blockchain, the network needs to inform John and Thomas that the transaction was made. The wallets are connected to some of the nodes, which will send a notification to both John and Thomas, that the payment was completed.

3.1.1 Transaction structure

The Bitcoin transaction structure is described below, shown in Figure 3.1, and is based on [1, 2, 11].

- The **version number** is used for backward compatibility and to distinguish between different types of transactions. For example, if the format of a transaction changes, a different version of the transaction may be used to ensure backward compatibility with previous versions. There are currently versions 1 and 2. Version 2 indicates that BIP 68 [15] applies. This proposal specifies a new way to control transaction validity. Essentially, it allows the user to set a time limit for using the transaction based on the time the previous transaction was created [8].

- **Flag** indicates the presence of witness data.

- **In-counter** specifies the number of inputs (also known as UTXO, Unspent Transaction Output) in the transaction. It is a variable length integer (VI), which means that it can be encoded in different ways depending on the number of inputs.

- **List of inputs**, where each Transaction input (Txin) consists of:
  - Previous transaction hash that contains the spendable output.
  - Previous txout-index of an array to identify the spendable output.
  - The length of the Txin script indicates the size of the locking script.
  - Txin scriptSig proves that the sender has the right to spend the bitcoins. Usually contains a digital signature that matches the locking script of the previous txout and public key, but that depends on the type of script being used. Script types are described more in Section 3.2.
  - The sequence number is used to determine the order in which transactions are added to the blockchain. For example, if a transaction has a sequence number of 0xFFFFFFFF, it can be included in a block as soon as possible, while

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Version                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Flag              |      In-counter (1 to 9B)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                List of inputs (variable length)               :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Out-counter (1 to 9B)                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                List of outputs (variable length)              :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               :
:                  Witnesses (variable length)                  :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Locktime                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

One tick mark represents one bit position, if there is not the
             length specified in the brackets.

Transaction input - Txin                    Transaction output - Txout
+--------+--------+--------+--------+       +--------+--------+--------+--------+
|  Previous Transaction Hash (32B)  |       |             Value (8B)            |
+--------+--------+--------+--------+       +--------+--------+--------+--------+
|       Previous Txout-index        |       |   Txout script length (1 to 9B)   |
+--------+--------+--------+--------+       +--------+--------+--------+--------+
|     Txin script length (1 to 9B)  |       |  Txout scriptPubKey (var. length) |
+--------+--------+--------+--------+       +--------+--------+--------+--------+
|  Txin scriptSig (variable length) |
+--------+--------+--------+--------+
|          Sequence number          |
+--------+--------+--------+--------+
```

Figure 3.1: Bitcoin transaction format (year 2023).

a transaction with a lower sequence number will be delayed until a specified block height or time has been reached. Also, with BIP 125, [16] feature Opt-in Full Replace-by-Fee Signaling was added. It enables spenders to include a signal in their transaction to indicate that they may want to replace the transaction with a new one at a later time. This signal gives users the flexibility to adjust the transaction fee in real-time, allowing them to increase the chances of their transaction being confirmed quickly.

- **Out-counter** specifies the number of outputs in the transaction. Like the In-counter, it is encoded as VI.
- **List of outputs**, where each Transaction output (Txout) consists of:
  - Value shows the amount of bitcoins which is sent to the receiver.
  - The length of the Txout script indicates the size of the locking script.

    - Txout scriptPubKey defines the list of instructions that the receiver must follow to spend the output in the future.

- **Witness** field is present only in SegWit transactions. It contains the information needed to verify the spending authorization for the transaction. Basically, it consists of signature scripts that are described in the following section.

- **Locktime** specifies the earliest time (in Unix time format) or block height at which this transaction can be added to the blockchain. Locktime enables signers to generate transactions that can only be executed in the future. This allows signers to change their mind until the transaction becomes valid. If any of the signers wish to modify the transaction, they can create a new transaction that does not use the locktime feature. The new transaction will use one of the same outputs as the locktime transaction, making the locktime transaction invalid if the new transaction is included in the blockchain before the time lock expires.

3.2 Scripts

Bitcoin scripts are used to define the conditions under which a transaction can be spent. Scripts are written in a simple stack-based language called Bitcoin Script, which is processed from left to right. They consist of two fields, which are data and opcodes. Data can be represented by public keys and digital signatures. Opcodes represent different operations, such as pushing data to the stack, popping data from the stack, performing arithmetic calculations, checking signatures, and verifying hashes.

Scripts are part of the transactions in the **scriptSig** and **scriptPubKey** fields. The scriptPubKey is the output script responsible for specifying the conditions that must be met to spend the bitcoins, such as providing a public key and a signature that matches a certain address. On the other hand, the scriptSig is the input script that provides the necessary data to satisfy the scriptPubKey, such as the public key and signature of the sender.

When a transaction is executed, the scriptSig and scriptPubKey are concatenated and executed by the Bitcoin clients to verify the validity of the transaction. If the execution results in a true value on the top of the stack, the transaction is considered valid and the bitcoins can be spent by the receiver. However, if the execution results in a false value, the transaction is considered invalid and the bitcoins remain unspent. There are different types of scripts in Bitcoin, each with its format and functionality [2, 11].

Addresses

A Bitcoin address is a unique identifier used to receive Bitcoin payments, similar to an email address used to receive emails. The address is publicly known to anyone who wants to send funds to it. The owner of the address is the only person who can access the funds using the private key associated with the address [21].

There are several address formats, depending on the type of script that is being used. The beginning of a Bitcoin address can often indicate the type of script it uses. Table 3.1 summarizes the scripts and addresses related to it.

The most common scripts are described below and are based on BIP 16 [3], BIP 141 [20], BIP 341 [29], Bitcoin Wiki [2] and learn me a bitcoin [28].

Script	Address starts with	Example
P2PKH	1	**1**KVzBuzuc28...
P2SH	3	**3**MxwgJLqN2k...
P2WPKH	bc1q	**bc1q**7jc2ql7...
P2WSH	bc1q	**bc1q**5laclf7...
P2TR	bc1p	**bc1p**ey6g6w9...

Table 3.1: Connection between Bitcoin scripts and addresses

3.2.1 Pay-to-Public-Key-Hash (P2PKH)

P2PKH is an improvement over the earlier P2PK (Pay-to-Public-Key) method. In P2PKH transactions, the recipient's Bitcoin address is a hashed version of their public key. This allows for greater security and privacy, as the public key is not revealed until the bitcoins are spent. The scriptPubKey contains the recipient's P2PKH address, which is a hashed version of their public key. The scriptSig contains a signature that proves that the sender has the private key corresponding to the P2PKH address. When the transaction is validated, the recipient can then use their private key to sign a message that proves that they have ownership of the public key corresponding to the P2PKH address. This signature is then included in the input script of a subsequent transaction to spend the bitcoins. P2PKH was introduced in the first version of Bitcoin, together with the P2PK script.

3.2.2 Pay-to-Script-Hash (P2SH)

P2SH allows for more complex locking scripts than the standard P2PK or P2PKH transactions. In a P2SH transaction, instead of including the public key hash or script in the scriptPubKey, a hash of the script is used. This hash is then included in the scriptSig, along with the full script that can unlock the bitcoins. The complete script that can unlock the bitcoins is called the redeem script. The redeem script is a script that the recipient provides, which corresponds to the hash in the scriptPubKey. P2SH was raised in April 2012 via BIP 16 [3].

3.2.3 Pay-to-Witness-Public-Key-Hash (P2WPKH)

P2WPKH was introduced with the Segregated Witness (SegWit) soft fork. The recipient's Bitcoin address is a hashed version of their public key, similar to P2PKH. However, the scriptPubKey contains a witness program instead of the recipient's P2PKH address. The witness program is a script that is used to validate the transaction and prove that the sender has the private key corresponding to the P2WPKH address. The scriptSig contains the signature that proves that the sender has the private key corresponding to the P2WPKH address [20].

3.2.4 Pay-to-Witness-Script-Hash (P2WSH)

P2WSH was introduced together with P2WPKH in SegWit. The main principle is the same as for P2SH, meaning that the hash of the script is included in the scriptPubKey and then the spender must provide the redeem script and signature to unlock the bitcoins. P2SH and P2WSH differ in terms of where the scriptSig content was previously placed and how the scriptPubKey is changed.

3.2.5 Pay-to-Taproot (P2TR)

P2TR is the newest script that was introduced in Bitcoin. It uses a concept called Merkle trees to combine multiple possible spending conditions into a single compact script. This means that a single Taproot output can represent many different types of transaction.

The way it works is that Taproot allows multiple spending conditions to be combined into a single Merkle tree. Each branch of the tree represents a different spending condition, such as a specific time delay, a certain signature, or a certain combination of signatures. When a transaction is made, the sender can choose which branch of the Merkle tree to use, depending on the specific spending condition they want to fulfill.

Instead of placing spending conditions directly in the scriptSig (like in previous scripts), these conditions are hashed and placed in a new field called the witness field. This witness field contains cryptographic proofs that the transaction meets the spending requirements defined in the scriptPubKey, which remains mostly unchanged.

The witness field includes a Merkle tree, but unlike the Merkle tree used in the script itself, this tree holds all the possible spending conditions. The entire Merkle tree is then hashed, and the resulting hash is stored in the witness field.

When validating the transaction, miners use the witness field to verify which specific spending condition was used to create the transaction. If the chosen branch is valid according to the scriptPubKey, the transaction is considered valid and can be added to the blockchain [29].

3.3 Propagation of transactions

The algorithm for propagating transactions in the Bitcoin network is based on broadcasting transactions to neighboring nodes in the peer-to-peer network. Then, the neighboring nodes send it to their peers, and this process repeats. Each node has rules that determine which transactions are valid and how to verify them. Furthermore, the node stores transactions in a memory pool (mempool) , which contains all transactions that the node receives. Transactions in mempool are considered unconfirmed and have not yet been included in a block. The propagation of transactions, initiated or received by a node, unfolds through the following process, and the main source of information for this section is based on the Bitcoin Wiki, Bitcoindeveloper P2P Network webpage and the Bitcoin Core Repository [2, 5, 9].

1. The node verifies whether the transaction that was received is valid according to its rules. If the transaction is valid, the node adds it to its **mempool** and marks it as unconfirmed. If the transaction is invalid, the node rejects it and does not send it further.

2. The node then sends the transaction to all its peers in the network using a message of type inventory **INV**, which contains the unique identifier of the transaction (txID).

3. When a node receives a message **INV** from its peer, it checks whether it already knows the transaction by its txid. If it knows, it ignores the message **INV**. If it does not know, it asks the neighbor to send the whole transaction using a message of type **GETDATA**.

4. When a node receives a message **GETDATA** from its peer, it sends the entire transaction using a message of type **TX**. In this way, the transaction is transferred between the nodes that have expressed interest in it.

5. When a node receives a message **TX** from its peer, it repeats the first two steps: it verifies the validity of the transaction and adds it to its **mempool**. Then it sends it to its other neighbors using a message **INV**.

This algorithm ensures that every node on the network learns about new transactions that meet its rules. Redundancy and unnecessary communication between peers is minimized, because each node sends only those transactions that other neighbors do not know. The process of propagating a received transaction is visualized as a Finite State Machine (FSM) in Figure 3.2.

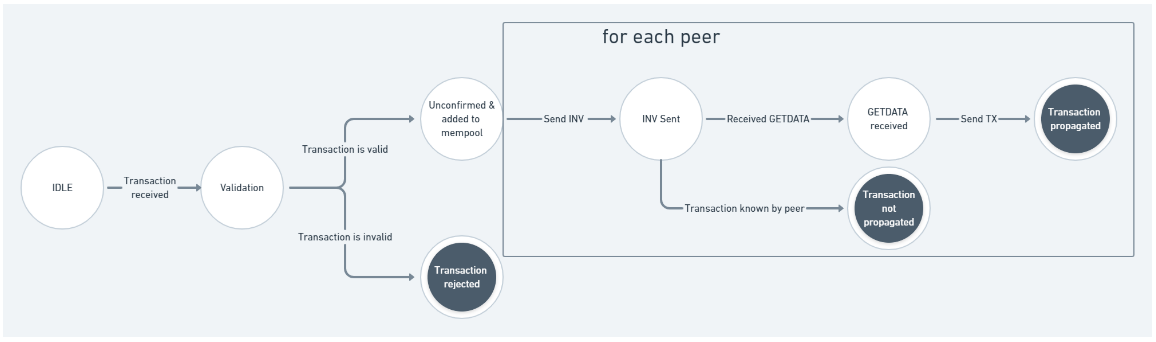


Figure 3.2: The propagation process of the received transaction by a Bitcoin node represented by a Finite State Machine.

Transaction propagation is associated with the concepts of outgoing and incoming connections. In the context of Bitcoin, the inbound and outbound connections are separated based on who initiated the connection between two nodes in the Bitcoin P2P network. An inbound connection is initiated by a remote peer who wants to connect to a node. In contrast, a node initiates an outbound connection to connect to a remote peer. Inbound and outbound connections are used to transmit and receive transactions and blocks.

By default, there is a limit on the number of connections that a Bitcoin node can maintain. This helps manage bandwidth usage and ensure smooth operation for most users. Bitcoin Core allows for a maximum of 125 total connections, where 11 can be outbound and the remaining 114 inbound. The number of inbound connections can be changed from the default value to a potentially unlimited number compared to outbound connections, where the default maximum values cannot be changed. Of the 11 outbound connections, a total of 8 are full-relay, which can be used to receive and transmit all types of data on the Bitcoin network, including unconfirmed transactions. The remaining three connections are used for purposes other than the transmission of transactions [6].

Both types of connection use an exponential distribution when it comes to transmitting the transactions. The exact distribution function `GetExponentialRand()` from the Bitcoin Core Repository [5] is shown in Listing 3.1. The difference is in the average relay interval. Outbound connections use an interval of 2 seconds for broadcasting the transactions, and inbound connections use 5 seconds.

```
/* Source for the delay:
    https://github.com/bitcoin/bitcoin/blob/2b260eadf7960290328e13dbdb029fd506105
    ca4/src/net_processing.cpp#L146 */
```

```
/* Average delay between trickled inventory transmissions for inbound peers. */
static constexpr auto INBOUND_INVENTORY_BROADCAST_INTERVAL{5s};
/* Average delay between trickled inventory transmissions for outbound peers. */
static constexpr auto OUTBOUND_INVENTORY_BROADCAST_INTERVAL{2s};

/* Source for the exp function:
    https://github.com/bitcoin/bitcoin/blob/2b260eadf7960290328e13dbdb029fd5061
    05ca4/src/random.cpp#L764 */
std::chrono::microseconds GetExponentialRand (std::chrono::microseconds now,
    std::chrono::seconds average_interval)
{
    double unscaled = -std::log1p(GetRand(uint64_t{1} << 48) *
        -0.0000000000000035527136788 /* -1/2^48 */);
    return now + std::chrono::duration_cast<std::chrono::microseconds>(unscaled *
        average_interval + 0.5us);
}
```

Listing 3.1: Bitcoin Core exponential function used by inbound/outbound connections to broadcast the transactions.

The term outbound/inbound connections can be used in conjunction with reachable/unreachable nodes. A node is considered reachable if it is capable of accepting incoming connections from other peers. Conversely, a node that is unable to accept incoming connections is deemed unreachable. A node may be considered unreachable due to being behind a firewall, connecting via a proxy, or hosting in a private network, typically behind a NAT (Network Address Translation) device.

As mentioned at the beginning of this section, unconfirmed transactions are stored in the mempool. There are various ways for a transaction to leave the mempool. The reasons are the following:

1. The transaction was inserted in a block.

2. The transaction expired by timeout. Every node has a mempool expiration time limit, after which transactions are removed from the mempool. By default, this time limit is set to 14 days. If a transaction remains unconfirmed for more than 14 days, it will be removed from the mempool.

3. The transaction was replaced. If a transaction has a low fee and is stuck in the mempool, it can be replaced with a new transaction that has a higher fee. This is made possible by the Replace-By-Fee (RBF) implementation. There are two variants of RBF, Full RBF, and Opt–in Full RBF. The difference between them is that with Full RBF any new transaction that consumes at least one of the same inputs as the original can replace it. Whereas Opt–in Full RBF requires the sender to explicitly mark the transaction as replaceable when creating it. For more information, see BIP 125 [16].

4. The mempool size has reached its maximum size limit and a new transaction with a higher fee is accepted. Transactions are sorted by fee per size in the mempool. Transactions with lower fees are the ones at the bottom of the mempool and can be evicted from the mempool. This means that the transaction at the bottom of the mempool will be removed, even if it has not been confirmed yet. This situation is called a purge.

3.4 Nonce

While generating a single SHA-256 hash can be computationally intensive, modern computers are powerful enough to produce such a hash in microseconds. A block can theoretically be constructed in a fraction of a second, with the speed of construction depending on the computer's power. However, the system would not be decentralized if the miner with the most powerful computer always won the construction of new blocks. Additionally, blocks would be constructed in seconds instead of the intended 10 minutes, and all 21 million Bitcoins would be mined in just a few months rather than over decades. To avoid this issue, Nakamoto introduced the concepts of difficulty and nonce, where difficulty refers to the amount of work required to build a block.

The difficulty is calculated from the formula:

$$\text{Difficulty} = \frac{\text{GenesisBlockHashValue}}{\text{CurrentTargetHashValue}} \tag{3.1}$$

The difficulty is greater when the target value is smaller (that is, more zeros in the first digits). An acceptable hash must be smaller than the target. The target value is the same for all Bitcoin clients. The difficulty adjusts itself every 2 016 blocks, based on the recent performance of the network, to keep the average time between new blocks at 10 minutes. When the hash rate of the Bitcoin network increases, the difficulty also increases.

The valid hash must begin with a certain number of zeros and must be lower than the current target hash value. If the hash does not meet these criteria, the hashing process continues by changing the Nonce number.

Example

The demonstration of guessing the correct Nonce is described in the following example. In this case, the required number of zeros at the beginning of the hash will be one, because the target hash is 0f8868e5a027a30d... The message will be „This is Bitcoin.“ and starting with Nonce 00000000. The Nonce is added to the message. The final SHA-256 hash of „This is Bitcoin. 00000000“ is ca31863c2cd26b00..., which does not begin with a zero. Then the nonce is increased by one, and the message is hashed again. It takes 6 loops to generate a lower hash starting with zero. The entire process is shown in the table 3.2. This section was based on the book Blockchain, Bitcoin, and the Digital Economy [21].

Table 3.2: Hashing with Nonce

Loop	Message	Nonce	Hash
1	This is Bitcoin.	00000000	ca31863c2cd26b00...
2	This is Bitcoin.	00000001	c9e3f09ca255f231...
3	This is Bitcoin.	00000002	7af606c351f058e9...
4	This is Bitcoin.	00000003	7ccd5a0b22f397a7...
5	This is Bitcoin.	00000004	8dc9f37ed0c77040...
6	This is Bitcoin.	00000005	0b2468e5a027a30d...

The hash generated in the 6th loop is lower than the target hash. In that case, the Nonce was guessed and a new target hash was created.

$$\begin{aligned} 0b2468e5a027a30d... &< 0f8868e5a027a30d... \\ NewTargetHash &= 0b2468e5a027a30d... \end{aligned} \tag{3.2}$$

Chapter 4

OMNeT++

OMNeT++ is a discrete-event simulation environment, and the name itself stands for the Objective Modular Network Testbed in C++. The main purpose is to model and simulate complex computer and communication systems. It is an open-source, component-based modular simulation framework that allows users to create simulations for various domains such as networking, wireless communication, validation of hardware architectures, and more. OMNeT++ supports a range of communication protocols and technologies, including TCP/IP, UDP, IP, Ethernet, and IEEE 802.11 wireless networks. It also supports the integration of external tools and libraries, such as MATLAB and NS-3. The bridge between the user and the simulation is provided by a graphical user interface (GUI) or the command line. The GUI is called the OMNeT++ IDE, which provides a variety of tools for creating, debugging, and visualizing simulation models. The IDE includes features such as a code editor, a graphical network editor, and a simulation runtime environment. The OMNeT++ Simulation Manual [27] serves as a source of information about OMNeT++.

In this thesis, the OMNeT simulator is used to create a model of a simplified Bitcoin client, that is able to create, send, and receive messages which represent Bitcoin transactions in the Bitcoin network. The design and the implementation of the model is described in next chapters.

Modeling

One of the central elements of the OMNeT++ infrastructure is a component architecture for simulation models. These models are constructed from reusable modules known as components. Modules at the lowest level of the module hierarchy are called simple modules, and they are programmed in the C++ language. Simple modules can be grouped into compound modules, and the whole model is called Network. The structure of a simulation model is described by the user in the Network Description (NED) language. NED lets the user specify simple modules and connect and assemble them together into compound modules.

Modules in an OMNeT++ model exchange messages to communicate. Messages from simple modules are commonly sent through gates, but can also be sent directly to the modules for which they are destined. Gates serve as the input and output interfaces of modules.

Chapter 5

Design of the simulation network

This chapter describes the three basic building blocks used to create the simulation. The first is the network topology, the second are the Bitcoin clients that represent the Bitcoin nodes, and the last are the monitoring nodes. Their purpose and behavior are described in the following subsections and the way they are implemented in the next Chapter 6.

Topology of the network

The topology consists of an arbitrary number of Bitcoin nodes, monitoring nodes, and the number of hops from the monitoring node to `bitcoinNode0` that generates the transaction. The topology structure is based on the user inputs given in the command line.

Bitcoin client

The Bitcoin Core client is the most widely used type of Bitcoin client in the world. More than 93 percent of the Bitcoin network is made up of Bitcoin Core clients, as shown in Appendix A [12]. However, the user agent is reported by the client, meaning if the client has its custom implementation, it can still report himself as Bitcoin Core agent but behave differently. The design of the Bitcoin client in this thesis is very simplified compared to the real Bitcoin Core client. Clients acting as nodes on the network can create, receive, and send messages representing Bitcoin transactions. In this thesis scenario, the message is created only by a `bitcoinNode0`. This message represents the transaction ID (txID) which is a unique identifier of the transaction on the network. The transmission behavior represents the INV messages from the Bitcoin Core Project [5]. Each client can have an unlimited number of inbound connections to other clients, and maximum of 8 outbound connections. Transactions are sent through connections based on the exponential function `GetExponentialRand()` described in Section 3.3. To remember if the transaction was already received, each node has its own mempool. The pseudocodes for creating and handling the received transaction shown in Listings 5.1 and 5.2 are used as a template for implementation.

Monitoring node

The monitoring node function in the topology is to simply collect information on when the transaction was received together with the neighbor name and txid. This behavior is ensured by connecting the monitoring node to all reachable Bitcoin nodes. Additionally, the monitoring node can also propagate messages as a Bitcoin client with the same propagation algorithm. The simulation ends when the monitoring node receives the transaction from all connections. The simulation output is a CSV file that contains all the records collected by the node. This file is then processed by an analytics script that displays the selected nodes of the simulation in terms of transaction propagation speed.

```
/* Function that generates a unique txID */
generateRandomTxID():
    /* Logic for creating a unique txID */
if nameOfTheNode = bitcoinNode0:
    txID ← generateRandomTxID()
/* Use the propagation algorithm now */
```

Listing 5.1: Pseudocode for creating a transaction.

```
OUTBOUND_INTERVAL ← 2 seconds
INBOUND_INTERVAL ← 5 seconds
if mempool.contains(receivedTx): n
    mempool.add(receivedTx)
    for all outbound connections:
        delay ← GetExponentialRand(OUTBOUND_INTERVAL)
        sendToPeer(delay) /* non-blocking function */
    delay ← GetExponentialRand(INBOUND_INTERVAL)
    for all inbound connections:
        sendToPeer(delay)
```

Listing 5.2: Pseudocode for the propagation algorithm.

Chapter 6

Implementation

This chapter describes the implementation and validation of a simulation that was developed based on the design of the previous Chapter 5. Parts of the programming code described in the text are highlighted in this way: `functions()`, `variables`, `structures`, `etc`. The general objective of the simulation is to monitor the transaction and create a CSV file with the captured results for later analysis. The simulation involves the propagation of INV messages from the Bitcoin Core Project [5]. Furthermore, monitoring nodes are present to track the time at which the transaction propagates from different nodes.

6.1 Simulation

This section dives into the implementation details of the simulation designed for the OMNeT++ simulator. The pipeline with the core parts of the simulation is shown in Figure 6.1. A class diagram representing the code structure for the simulation is shown in Figure 6.2. The following subsections describe the simulation parameters and its topology, the behavior of the modules, how the transactions are created and transmitted, the creation of the CSV file, and deviations from the real Bitcoin network.

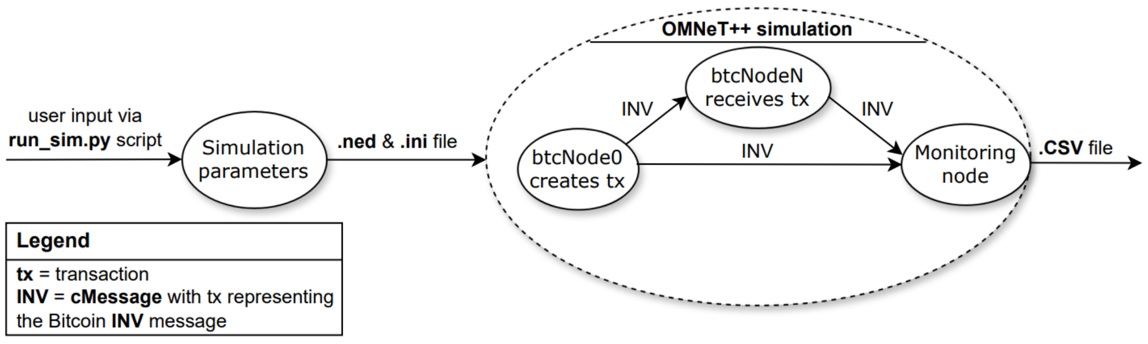


Figure 6.1: The pipeline with the core parts of the simulation with their inputs and outputs.

6.1.1 Simulation parameters

Before the simulation is run, the network topology must be specified. The topology is generated based on the `BitcoinNetwork.ned` file. This file is created together with the `omnetpp.ini` file within the `run_sim.py` script, which executes the simulation, and its implementation is described in more detail in Section 6.2. To run the simulation with all the mentioned files, the Python script must be run from the `/simulations` folder.

The execution command sequence is `python3 run_sim.py <1> <2> <3> <4> <5> <6>` where the numbers are represented by the following parameters.

1. `number of Bitcoin nodes`
2. `number of monitoring nodes`
3. `hop distance` described in the next paragraph
4. `number of simulation runs`
5. `number indicating the type of simulation` where 0 is for GUI and 1 is for command line interface (CLI) simulation
6. `number indicating if monitoring node should propagate messages` where 1 means propagate and 0 do not propagate

The `hop distance` indicates the number of nodes through which the transaction must pass from the source to reach the destination, where in this scenario the source is `bitcoinNode0` and the destination is the `monitoringNode0`. For example, if the hop distance is 0, the `bitcoinNode0` and the `monitoringNode0` are directly connected, because the `bitcoinNode0` accepts the inbound connections. If the hop distance is 1, then it means that there is 1 other node between `bitcoinNode0` and the `monitoringNode0`, because the `bitcoinNode0` does not accept any inbound connections and makes only outbound connections. For clarification, see the generated topology in Figure 6.3, with 5 bitcoin nodes, 1 monitoring node, and a hop distance of 1.

The `number of simulation runs` parameter specifies how many times the simulation will run with the same topology. The `seed-set` variable is used to achieve different results even if the topology is the same. This variable is part of the `omnetpp.ini` file that is created by the `run_sim.py` script. The value of the `seed-set` is based on the repeat loop counter variable `$repetition`, which represents the current run number. `Seed-set` and `$repetition` are described in detail in the OMNeT++ manual in Section 10.4.6 named Repeating Runs with Different Seeds [27].

6.1.2 Creating a transaction

The simulation begins with the initialization of the `BitcoinNode` class. The `bitcoinNode0` instance creates a new cMessage requesting a new transaction to be generated. The messages are handled by the `handleMessage(cMessage *msg)` function. There are two types of messages, self–messages and messages from other nodes. If the message is a self–message, the function `generateAndBroadcastTransaction()` is called. This function generates a transaction and calls the `broadcastMessage(cMessage *msg)` function of the `NodeBase` class to broadcast it to outbound and inbound connections. The transaction is created and identified by a 64–bit hexadecimal number and does not contain any other information. The transaction is generated by the `generateRandomTxID()` function. This function generates random characters that are concatenated to form a txID. The character is chosen using a function charset[`intuniform(0, charsetSize - 1)`], which generates a uniformly distributed random integer. Finally, the function returns the generated `txID` string. This method of generating a txID is not how Bitcoin transaction IDs are actually created on

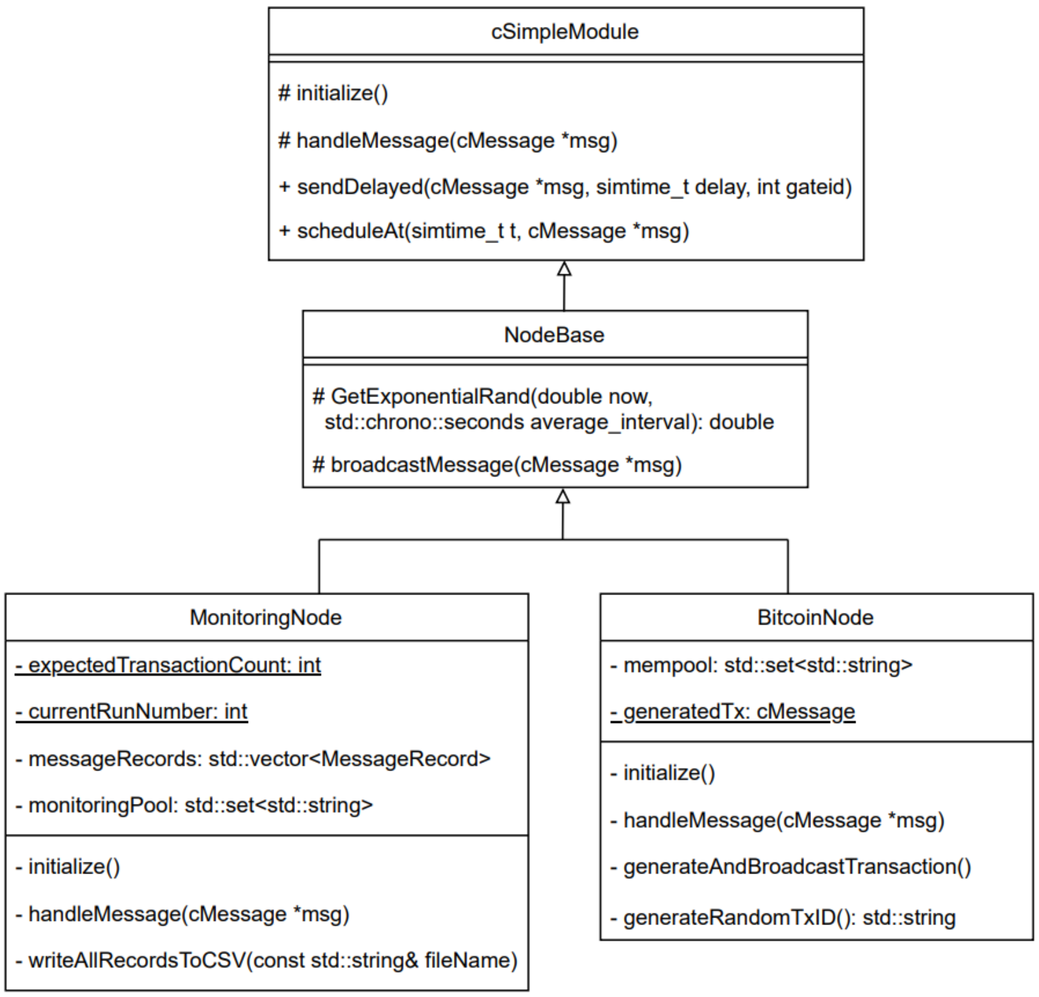

cSimpleModule
# initialize()
# handleMessage(cMessage *msg)
+ sendDelayed(cMessage *msg, simtime_t delay, int gateid)
+ scheduleAt(simtime_t t, cMessage *msg)
NodeBase
# GetExponentialRand(double now, std::chrono::seconds average_interval): double
# broadcastMessage(cMessage *msg)
MonitoringNode
- expectedTransactionCount: int
- currentRunNumber: int
- messageRecords: std::vector<MessageRecord>
- monitoringPool: std::set<std::string>
- initialize()
- handleMessage(cMessage *msg)
- writeAllRecordsToCSV(const std::string& fileName)
BitcoinNode
- mempool: std::set<std::string>
- generatedTx: cMessage
- initialize()
- handleMessage(cMessage *msg)
- generateAndBroadcastTransaction()
- generateRandomTxID(): std::string

Figure 6.2: The `BitcoinNode` class represents individual nodes in the Bitcoin network simulation. All attributes and methods are used only within the `BitcoinNode` class and therefore are implemented as private. The same remains for the `MonitoringNode` class which is responsible for monitoring, recording, and exporting message records. The `NodeBase` class contains protected methods that provide delay calculation and transaction propagation, which together form the resulting propagation algorithm. The `MonitoringNode` and `BitcoinNode` classes inherit from the `NodeBase` class to access these functions. The `cSimpleModule` class is the OMNeT++ built-in class that provides methods for sending messages to other modules, scheduling events, and performing other simulation-related tasks. It is the core OMNeT++ class from which other implemented classes inherit to define the specific simulation behavior.

the Bitcoin network. In practice, txIDs are hash values of transaction data. Once the transaction is created, it is added to the node `mempool` and sent out to the neighboring nodes.

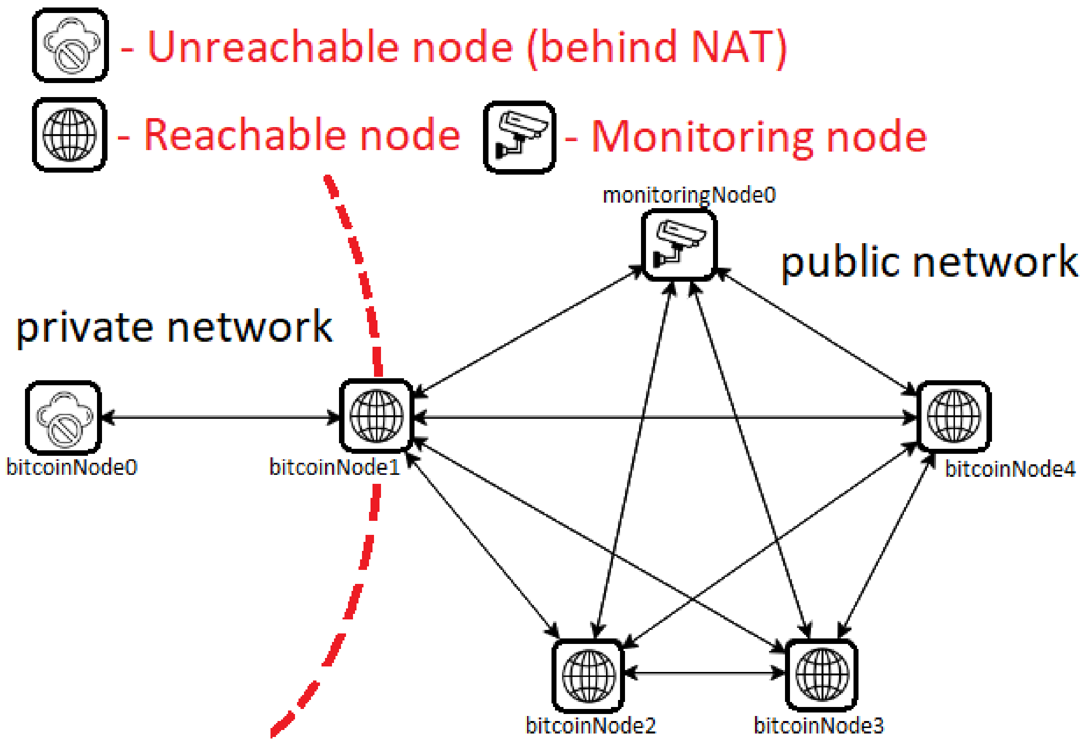


Figure 6.3: Network topology from the OMNeT++ generated by the number of 5 bitcoin nodes, 1 monitoring node, and a hop distance of 1. The `bitcoinNode0` is an unreachable node because it is behind NAT in the private network and thus does not accept any inbound connections. NAT device is between the `bitcoinNode0` and `bitcoinNode1`. The `bitcoinN-ode0` creates outbound connection to random Bitcoin node which is now `bitcoinNode1`. The monitoring node on the top of the topology is connected through outbound connections to all reachable nodes.

6.1.3 Receiving a transaction

When the transaction is received from another node, it first gets the TXID of the transaction. Then it checks if the transaction is in the `mempool`. The `mempool` is implemented as a std::set class from the C++ Standard Template Library (STL). The main reason for choosing the set was that it ensures that each txID is unique within the mempool. The `re-ceivedTxID` is inserted into the `mempool` if it is not already there. Otherwise, `receivedTxID` is known and thus ignored. The `cMessage *msg` containing the unknown transaction is then passed as a parameter to the `broadcastMessage()` function of the `NodeBase` class. This function is used to broadcast transactions to all inbound and outbound connections. Before that, the simulation time needs to be converted to microseconds, in which the delay is calculated. The calculation of delay is performed in two for loops, which iterate based on the number of outbound/inbound connections. The same exponential distribution and outbound/inbound intervals are used for the relay with the `GetExponentialRand(double now, std::chrono::seconds average_interval)` function as shown in Listing 3.1. The function is called only once for the inbound connections, but separately for each outbound connection. After the delay is calculated, it must be converted back to the simulation time before sending it. The non-blocking `sendDelayed()` function of OMNeT++ is used to send the transaction to other nodes through outbound/inbound connections, where neighboring nodes can be either Bitcoin nodes or Monitoring nodes.

6.1.4 Monitoring nodes

The monitoring nodes are connected through outbound connections to all Bitcoin nodes that accept inbound connections (those that are not behind NAT, firewall, or proxy, as mentioned in Section 3.3). The number of outbound connections is unlimited compared to the Bitcoin nodes, which can have a maximum of 8 outbound full-relay connections, as defined in the Bitcoin Core Project [6] and mentioned in previous chapters.

The monitoring node needs to keep track of the current simulation run number, which is later used in the analysis to distinguish from which run the results are from. The monitoring node has implemented a `MessageRecord` structure that stores three basic pieces of information. Transaction ID, the name of the neighbor from which the transaction was received, and a timestamp. The `MessageRecord` is created every time the monitoring node receives a message and is consequently pushed to the vector containing all records.

As with Bitcoin nodes, the `handleMessage(cMessage *msg)` function is responsible for the processing of messages. The behavior of the monitoring nodes can be set in two ways, either it will propagate transactions like Bitcoin nodes using the `broadcastMessage` function or it will not. The behavior is based on the `propagateMessages` parameter inside the `MonitoringNode.ned` file.

6.1.5 CSV file generation

The simulation ends when the instance of `monitoringNode0` receives the transaction from all nodes that propagate the transactions. After this is verified, the `writeAllRecordsToCSV-(„messageRecords.csv")` function is called. This function is responsible for writing all the accumulated `messageRecords` to a CSV file named `messageRecords.csv`. The function opens a CSV file for writing and if the file is empty (identified by the `tellp()` function that returns the current position of the put pointer), the header is written in the CSV file. The header consists of columns named `run, TXID, peer,` and `timestamp` and their purpose is described in the following.

- `run` - number of the simulation run. The simulation is repeated several times to obtain meaningful results. In that case, the run number needs to be tracked to differ the collected data.
- `TXID` - Transaction ID representing unique identifier of the transaction in the Bitcoin network.
- `peer` - name of the neighbor who sent the transaction to the monitoring node
- `timestamp` - time in seconds at which the transaction was received by the monitoring node

After the headers are written, the program iterates through each record stored in the `messageRecords` vector and writes its contents to the CSV file. Once all records have been written, the simulation is complete and ends. The `messageRecords.csv` file is later analyzed by the analysis script described in Section 6.3.

The simulation code must first be compiled using the `make` command from the `btc_-network` folder. The program source files are contained in the `/src` folder and the generated `messageRecords.csv` file can be found in the `/simulations` folder.

6.1.6 Deviations from real Bitcoin network

It should be taken into account that the simulation was performed on a specific topology, which may not exactly correspond to the real Bitcoin network, but its main goal was to implement the most similar transaction propagation behavior of the inventory messages as its implemented in the Bitcoin Core client. In addition, the real Bitcoin network consists of multiple client types with different parameters, making a significant portion of the Bitcoin network behave differently than the simulated one. Another point which must be taken into consideration is that all Bitcoin nodes in the simulation utilize the maximum number of connections with other peers, where for outbound connections it is 8 and for inbound it is unlimited.

6.2 Initialization script

The `run_sim.py` script is a Python script designed to automate the simulation of a Bitcoin network using the OMNeT++ discrete-event simulator. It generates `.ned` and `.ini` files for the simulation, executes the simulation using the provided parameters, and displays the output of the simulation. The script takes command-line arguments as input specifying the simulation structure as described in the beginning of the previous Section 6.1.1.

The script starts by parsing and checking the arguments provided via the command line. If the arguments are incorrect, a usage message or an error message is printed to provide the correct parameters. In order to simulate a network with a lower number of Bitcoin nodes than occurs in a real Bitcoin network, it is necessary to scale the maximum number of possible outbound connections accordingly. An equation with exponential behavior is used for scaling, and the result is obtained through the Newton-Raphson numerical method[1].

After initial validation and calculation, the generation of the `BitcoinNetwork.ned` file representing the network topology is started. Firstly, the `bitcoinNode` and `monitoringN-ode` submodules are created, where the `monitoringNode` submodules contains the `propa-gateMessages` and `monitoringNodes` parameters taken from the CLI input. These parameters are used in the simulation as described in the previous section. Once the submodules are created, it is possible to create connections between them to allow bidirectional communication.

The monitoring nodes and the Bitcoin nodes use gates for communication with other modules of the simulation. Each module has implemented one input gate `in[]` and two output gates `inbound[]` and `outbound[]`. First, connections between the monitoring nodes and the Bitcoin nodes are created. The script iterates over each monitoring node (represented by `num_monitoring_nodes` variable) and connects it to all `bitcoinNodes` beyond the hop distance. All connections from `monitoringNode` are from an `outbound[]` gate to the `bitcoinNode in[]` gate and bidirectional connections are made through an IdealChannel. This process is used in a similar way for all other connections. The code with the setup of the connection between the `bitcoinNodes` and `monitoringNodess` is shown in Listing 6.1.

[1] https://personal.math.ubc.ca/~anstee/math104/newtonmethod.pdf

```
# Generate connections between MonitoringNodes and BitcoinNodes
for i in range(num_monitoring_nodes):
    for j in range(hop_distance, num_bitcoin_nodes):
        ned_content += f"""
        monitoringNode{i}.outbound++ --> IdealChannel --> bitcoinNode{j}.in++;
        bitcoinNode{j}.inbound++ --> IdealChannel --> monitoringNode{i}.in++;
"""
```

Listing 6.1: Connection setup between `monitoringNode`s and `bitcoinNode`s.

Next, the outbound/inbound connections are generated between `bitcoinNode` submodules. It begins by initializing a `set()` named `established_connections`, aimed at tracking already established connections. Iterations are performed for each `bitcoinNode` based on the maximum number of outbound connections. For each node, a neighbor number is generated randomly to which it will establish an outbound connection. This random number starts at the `hop_distance` value to avoid creating connections to nodes that are behind NAT. The next step is to check if the generated number together with the current `bitcoinNode` is not already present in the `established_connections` set to avoid duplicate connections. If the connection is not there, it is created, added to `ned_content` and to the record of established connections.

After setting up connections between Bitcoin clients, the script proceeds to create the last type of connection, which is between the monitoring nodes. Iteration is performed on each pair of monitoring nodes and then the establishment of bidirectional connections between them. After generating all connection definitions, the `ned_content` is written in the newly created `BitcoinNetwork.ned` file and the topology of the network is ready.

The simulation needs to contain the configuration file, which is represented by `omnetpp.ini`. This file is generated with the following settings:

- network = BitcoinNetwork: Specifies the network model to be used in the simulation.
- repeat = `num_runs`: Specifies the number of times the simulation should be repeated.
- seed-set = $repetition: Specifies the seed value for random number generation.
- record-eventlog = false, cmdenv-performance-display = false, cmdenv-express-mode = true: Settings to speed up the simulation.

Once all files are prepared, the simulation can be run. Execution is based on the simulation type specified by the user (`sim_type` variable), according to which the script constructs a command to execute the simulation. The `run()` function from the `subprocess.py` module is used to run the simulation, capture the output, and print the message if the simulation was successfully performed or not.

6.3 Analysis script

The main purpose of the analysis script `analyze.py` located in the `/simulations` folder is to find out if some nodes propagate a transaction faster than others in most runs and thus occur at the top positions of the `messageRecords.csv` file. Such nodes may be potential source nodes for the transaction. Another purpose is to determine the average propagation time of a transaction or to compare two files in this aspect. The results are then visualized with graphs and saved as a png file. The usage command is `python3 analyze.py first_file top_nodes [-compare second_file]`, where both files represent the path to the

CSV files that should be analyzed and `top_nodes` parameter indicates the number of nodes that should be plotted. The `second_file` parameter is optional.

The script utilizes the pandas library for data manipulation and the matplotlib.pyplot library for plotting graphs. The argparse library is used for parsing the input parameters. The script first loads the `first_file` file into a pandas DataFrame called `data`. After loading the data, the code is split into three parts, where the nodes are calculated based on position, secondly based on timestamp, and in final the transaction propagation time is calculated.

Calculation based on position

The calculation based on position starts by adding a position number to each peer for each run in a new column called `position` in the DataFrame. Then, the calculation of `averagePosition` for each peer across all runs is done by grouping the `position` column for each peer and applying the `mean()` function to it. At the end of this part, based on the number of `top_nodes` parameter, nodes with the lowest `averagePosition` are selected for later plotting.

Calculation based on timestamp

The second part deals with the calculation based on timestamp. Firstly, the computation of the `timestamp_difference` between the first and other neighbors for each run are done. After that, the `averagePropagationTimePerNode` is calculated by grouping the `times-tamp_difference` for each peer and applying the `mean()` function as in the first part. Finally, nodes with lowest values are selected. The selected values from both parts are plotted as a bar graph, displayed, and saved as the `fastestNodes.png` file.

Calculation of the transaction propagation time

The final part handles the transaction propagation time. The number of simulation runs are extracted into `numberOfRuns` variable first. After that, the script selects the highest timestamp of each run in `maxTimestampPerRun` which is used as the values and indexes for the graph. In addition, the `averagePropagationTime` is calculated simply by summing up each maximum timestamp for each run and dividing it by the `numberOfRuns`. The `averagePropagationTime` value is displayed together with the transaction propagation time graph and saved as `txPropagationTime.png`. If `second_file` is passed as an input argument, the same principle is used and the graph is plotted with both selected data. Graph visualizations are included in Chapter 7.

6.4 Verification and validation

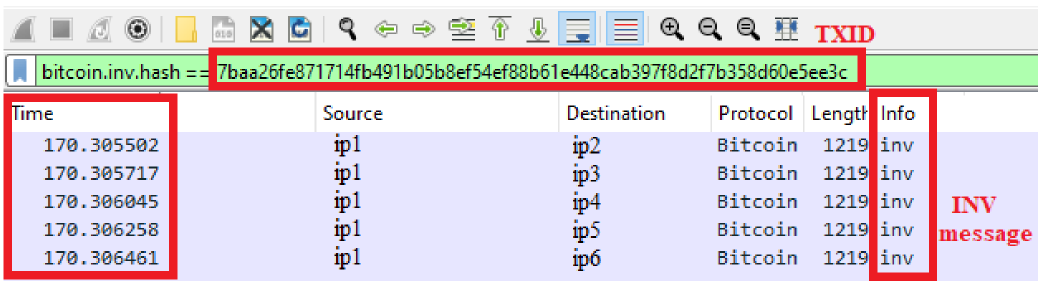
Verification of the simulation model, specifically verification of the exponential function `GetExponentialRand()` used to calculate the delay for message propagation, was tested by calling the function in a loop million times for each outbound and inbound interval. The average was calculated after the loop ended, giving the results corresponding to both a five-second inbound interval and a two-second outbound interval. This verification was performed on a total of 10 nodes with the results shown in Table 6.1.

The validation of the simulation model was based on a pcap file containing captured events from the Bitcoin network along with a listing of connected peers and their type of con-

Node	Average result in seconds for	
	outbound	inbound
1	2.00084	5.01009
2	2.00177	5.00937
3	1.99826	4.99002
4	1.9976	4.99734
5	1.99733	5.00579
6	1.99826	5.00326
7	2.00013	5.00851
8	1.99694	4.99525
9	2.00037	5.00229
10	2.00001	5.00218

Table 6.1: Verification results of the `GetExponentialRand()` function performed on a total of 10 nodes. The results correspond to the inbound and outbound interval values from the Bitcoin Core Project.

nection towards the source node running the Bitcoin Core client. The pcap originates from the source node, whose IP address is labeled as `ip1`. IP addresses of other nodes are labeled as `ip2, ip3, etc.` The pcap contains a transaction that was generated by the source node. The TXID of the generated transaction in reverse byte order that is used externally when searching for transactions on the internet is `3ceee5608d357b2f8d7f39ab8c441eb688ef54e-fb8051b49fb141787fe26aa7b`. To find the transaction in the pcap, the TXID needs to be in natural byte order, which is `7baa26fe871714fb491b05b8ef54ef88b61e448cab397f8d2f7b-358d60e5ee3c`. This TXID and the raw transaction data taken from mempool.space[2] were filtered in Wireshark to find the transaction and its propagation. The transaction was sent to all the inbound nodes at nearly identical times, as shown in Figure 6.4. This validates the information that for inbound peers, the exponential function is calculated only once for all of them.



```
bitcoin.inv.hash == 7baa26fe871714fb491b05b8ef54ef88b61e448cab397f8d2f7b358d60e5ee3c

Time            Source   Destination   Protocol   Length  Info
   170.305502    ip1      ip2           Bitcoin    1219    inv
   170.305717    ip1      ip3           Bitcoin    1219    inv
   170.306045    ip1      ip4           Bitcoin    1219    inv
   170.306258    ip1      ip5           Bitcoin    1219    inv
   170.306461    ip1      ip6           Bitcoin    1219    inv
```

Figure 6.4: Validation that Bitcoin clients use the exponential distribution function only once for all inbound connections since the time of the propagation from the pcap is almost identical.

The complete propagation process of the generated transaction for inbound peers from the captured pcap is shown in the sequence diagram in Figure 6.5. The `INV` and `getdata` messages were found based on the TXID and the `tx` message by the raw transaction data.

[2] `https://mempool.space/`

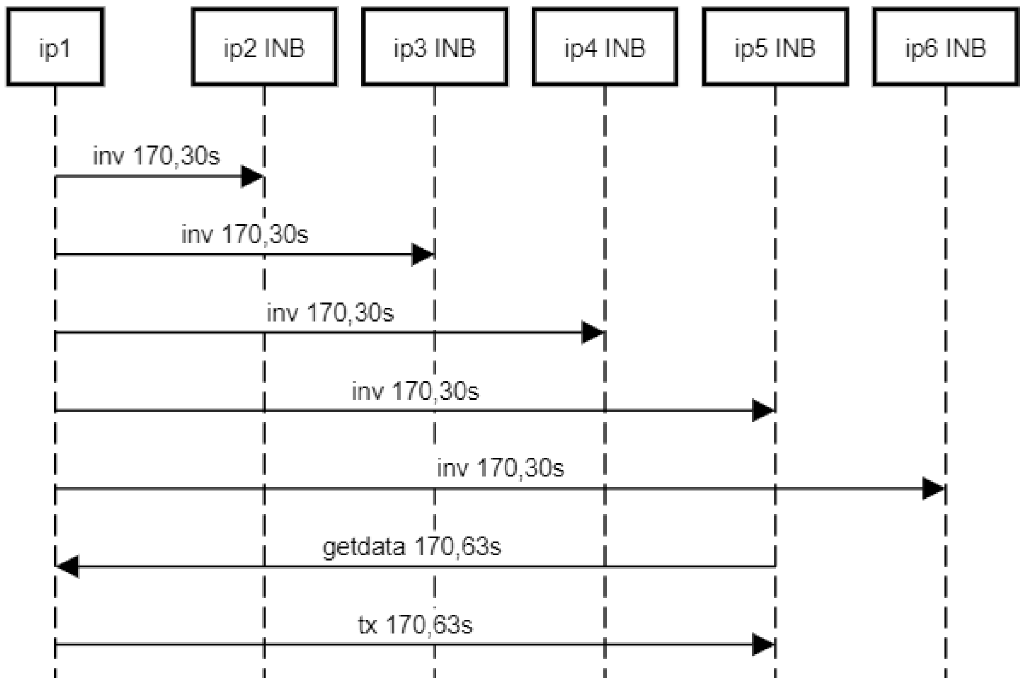


Figure 6.5: Sequence diagram of the captured process of the generated transaction for an inbound connections based on the pcap file containing Bitcoin network traffic.

It was also validated that for outbound connections, the transaction is sent at different times for each outbound peer. This indicates the use of an exponential function separately for each peer. The complete captured propagation process of the generated transaction for outbound peers is visualized in the sequence diagram in Figure 6.6. A comparison of `INV` messages for inbound and outbound connections shows that `INV` messages for outbound connections are propagated earlier because they have a lower broadcast interval. In the sequence diagram for outbound connections, it can be seen that some peers received the `tx` message even though they did not request it by a `getdata` message. This behavior is possible if the source node uses the implementation of the bitcoinj[3] library. If such a node creates a transaction, it can send `unsolicited tx` messages [10]. This behavior was noticed in the Bitcoin Core client from the reviewed pcap.

After verification and validation, it can be stated that the implemented Bitcoin client model works as expected. The implemented algorithm for propagating inventory messages works the same as in the real Bitcoin network, which has been verified on captured Bitcoin network traffic. The model lacks the propagation of `getdata` and `tx` messages compared to the real Bitcoin client. However, this missing implementation should not matter for obtaining results on the propagation of unconfirmed transactions, as the propagation of `INV` messages is sufficient. Another difference is that only one and the same node generate a transaction in the simulation model, whereas in the real Bitcoin network it could be all nodes, but again this should not affect the results, because each transaction is propagated by the same algorithm no matter what node generates it.

[3]https://bitcoinj.org/

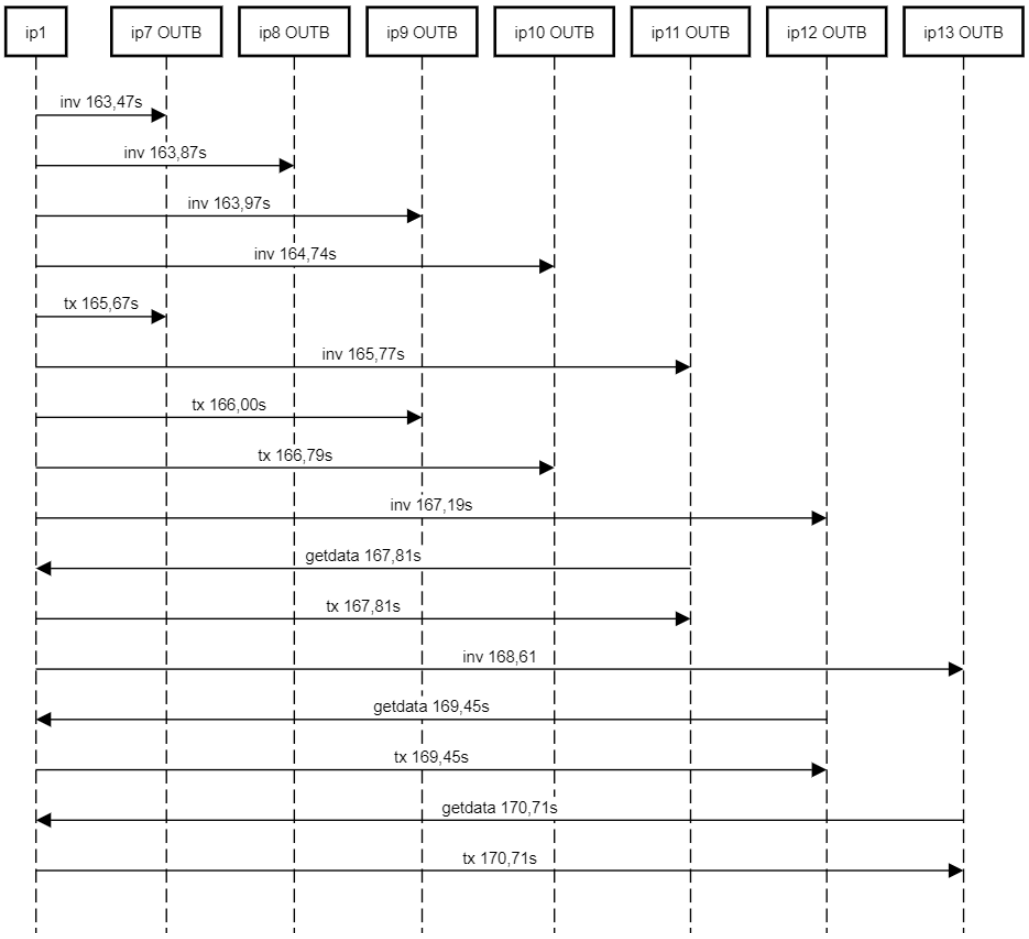


Figure 6.6: Sequence diagram of the captured process of the generated transaction for an outbound connections based on the pcap file containing Bitcoin network traffic.

Chapter 7

Analysis

This chapter describes the analysis of the results from the created simulation, specifically, what the results were and how they were obtained. As described at the end of Section 6.1, the created simulation differs from the real Bitcoin network in some ways. In other words, the results cannot be considered identical to those that would be obtained from a real Bitcoin network.

7.1 Identification of the possible source node of a transaction

The procedure for obtaining the results was to run the simulation multiple times, where the monitoring node recorded the time it received a transaction from which node. The main objective of the simulation was to try to identify a possible source node of the transaction. The `analyze.py` script described in Section 6.3 was used for this purpose. The possible source node is considered to be a node that, in most cases, propagates a transaction in a faster time than others and thus is at the top positions of the `messsageRecords.csv` file in most simulation runs. If such a node is identified, the question arises whether it is the source node of the transaction, but there are also other possibilities to consider. It could be a node that forwards transactions quickly because it has a custom implementation. Alternatively, it might be a node that has an inbound connection with the source node, especially when the source node is behind NAT. Since this thesis only considers honest Bitcoin Core clients without custom implementations, an identified node will be considered a source node or a node that is close to it.

Two types of metric were used for the analysis and it was the position and timestamp at which the monitoring node received the broadcast transaction. The plotted results are always the average of the captured values from all runs of a given network topology. In general, the lower the value on the y–axis representing the average position or timestamp, the higher the probability that the node is the source of the transaction. The bar graphs generated by the `analyze.py` script that was executed over the results obtained from the initialization `run_sim.py` script are shown in Figure 7.1. The graphs belong to a simulation that was run with parameters of 2000 Bitcoin nodes, 1 monitoring node, 0 hop distance, 100 simulation runs, CLI simulation, and monitoring node do not propagate messages. The monitoring node in this scenario is connected to all Bitcoin nodes in the network. The fastest node identified is `bitcoinNode0` and can be considered as a possible source node of the transaction.

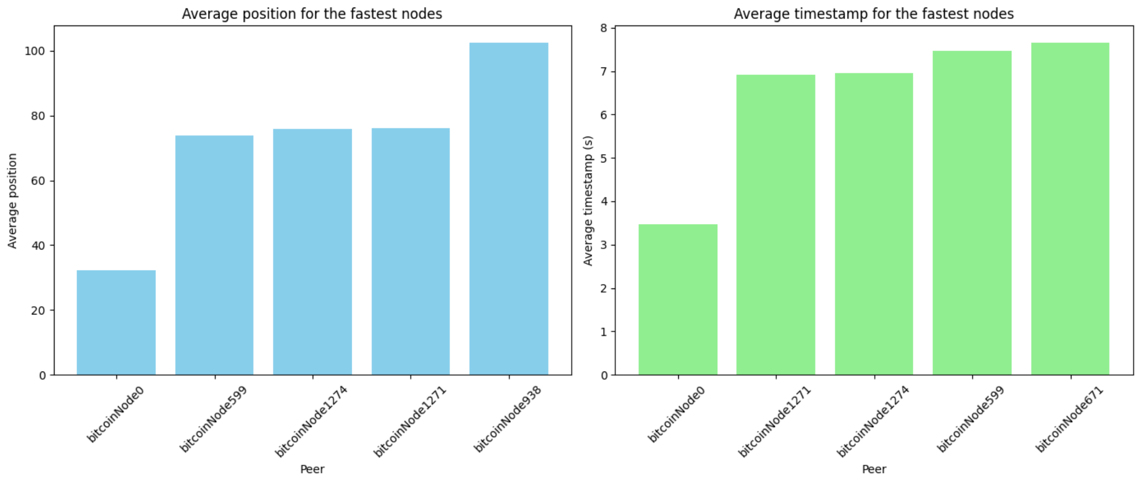


Figure 7.1: Identification of the possible source nodes based on the position and timestamp metric. The fastest node is `bitcoinNode0`, where the difference from the others is relatively large. In that case, this node can be considered as a possible source node of the transaction. The results are from simulation where the `bitcoinNode0` was directly connected to the `monitoringNode` with an inbound connection.

The analysis results are different when the source node is hidden behind NAT. The graphs in Figure 7.2 belong to a simulation that was run with parameters of 1000 Bitcoin nodes, 1 monitoring node, 1 hop distance, 1000 simulation runs, CLI simulation, and monitoring node do not propagate messages. The source of the generated transaction is `bitcoinNode0`, but it is behind the NAT device, which means that the monitoring node is not connected to it. Both graphs point to the same four nodes with approximate results. In that case, these nodes can be considered as the nodes that are close to the source node but cannot be considered as possible source nodes.

The simulation was run several times with different numbers of Bitcoin nodes (1000, 2000, 5000, 10000) and other parameters in various combinations. From the results of all simulations it was possible to determine that it was always possible to identify the source node if it was not located behind NAT and connected with the `monitoringNode` through an inbound connection, which means the simulation parameter of the hop distance was set to 0. Since the source node that generated the transactions was always `bitcoinNode0`, it is possible to state that the fastest identified node in this particular case was always the correct one. The results do not differ in such a large way from other nodes but in some small way, which means that it can still be considered as a possible source of a transaction. This means that if the entire Bitcoin network consisted of honest Bitcoin Core nodes and the source node of a transaction was not hidden behind NAT, it would be possible to identify the source node of the transaction. This means that actually, thanks to nodes with custom or different implementations, more anonymity can be maintained in the Bitcoin network.

Conversely, if the source node is behind NAT and is therefore not connected with the `monitoringNode`, the identification of it is essentially impossible. The analysis script is only able to identify the closest possible nodes to the source node that accepts inbound connections. This means that if the source node is hidden behind one NAT device, the script identifies the nodes to which the source node connects via its outbound gate, since the monitoring node does not know that the source node even exists when it is not connected to it.

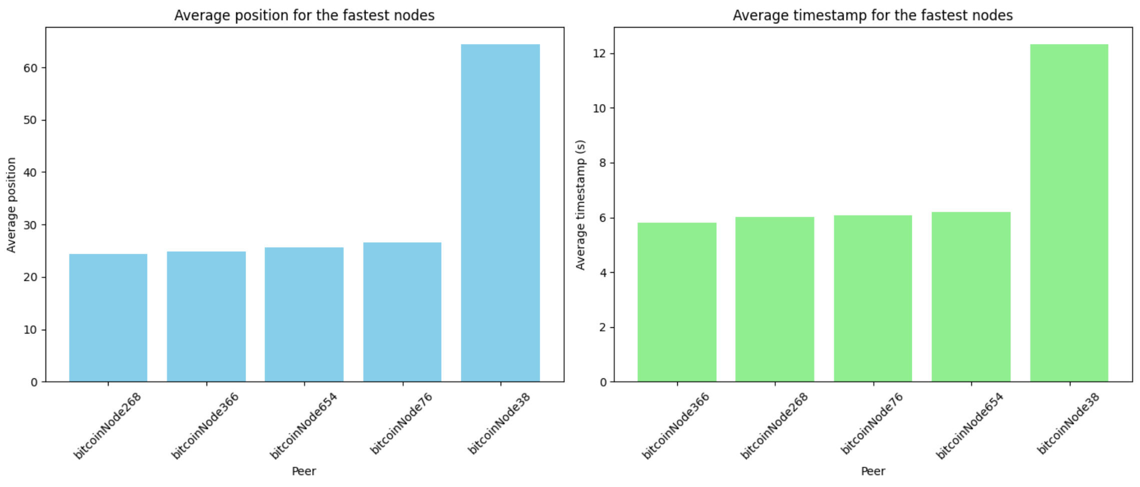


Figure 7.2: Identification of the possible source nodes based on the position and timestamp metric. The fastest node is `bitcoinNode1`, where the difference from the others is relatively huge. In that case, this node can be considered as a possible source node of the transaction or at least the node which is close to it.

7.2 Impact of monitoring nodes

Another result obtained is related to the monitoring nodes, specifically to the adjustable parameter `propagateMessages`. When the monitoring nodes are set to propagate messages, the messages spread throughout the network faster, and the entire simulation is completed in a faster time. This means that monitoring nodes can enable faster propagation of transactions, and thus can affect the network in some way. The comparison between the simulation with the monitoring nodes that propagate transactions and the topology where the monitoring nodes do not propagate them is handled by the `analyze.py` script. The output graph in Figure 7.3 shows the time for each simulation run in which the transaction was received by the monitoring node from all nodes on the Bitcoin network. The graph also shows the average transaction propagation time for both files, which is calculated from all runs. This value proves that transactions are propagated faster in the network if the monitoring nodes propagate transactions as well as Bitcoin clients. The graph belongs to a simulation that was run with parameters of 2000 Bitcoin nodes, 1 monitoring node, 0 hop distance, 100 simulation runs, CLI simulation, and monitoring node do not propagate messages and the same for the second file except the propagate messages parameter was set. From further running analyzes, it was evaluated that the more monitoring nodes in the topology, the faster the transaction propagated throughout the network. This fact is indicated by Table 7.1 containing the number of monitoring nodes that propagated the transaction along with the average time that the transaction was received from all nodes. The results are taken from a simulation that was run 100 times with 2000 Bitcoin nodes and a hop distance of 0.

Another finding is that if there are multiple monitoring nodes in the network that propagate transactions when the source node is behind NAT or even it is not, these monitoring nodes appear in the analysis script results, as shown in Figure 7.4 and 7.5. This means that if any of the monitoring nodes could send transactions immediately and not by the Bitcoin Core propagation algorithm, it could theoretically pretend to be the possible source node of the transaction, even though it is not.

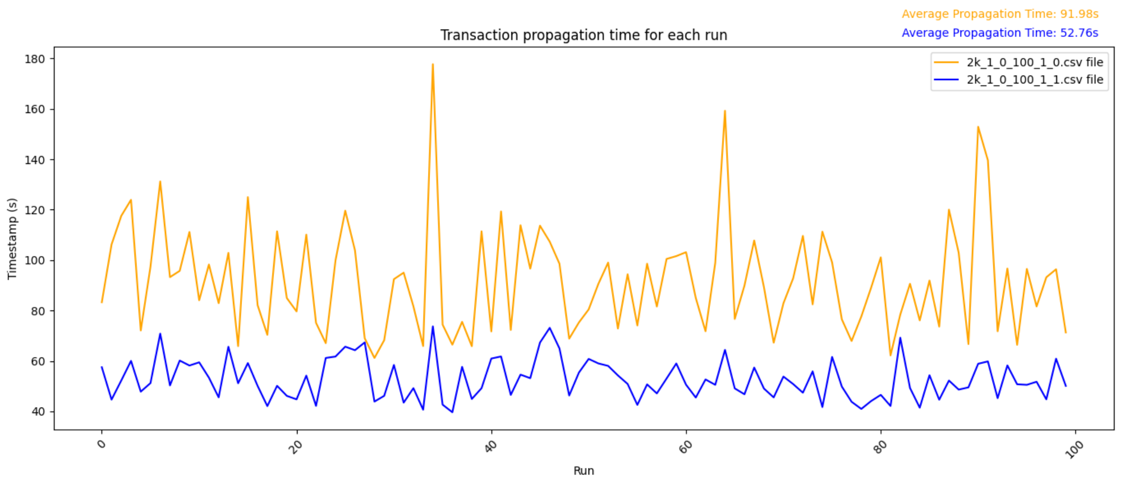


Figure 7.3: Comparison between the simulation with the monitoring nodes that propagate transactions (blue record) and the simulation where the monitoring nodes do not propagate (orange record) them. The comparison shows a significant speedup in transaction propagation when the monitoring nodes propagate them.

Number of monitoring nodes	Average propagation time
1	52.76s
2	50.72s
3	49.09s
4	48.52s

Table 7.1: Indication that monitoring nodes affect the propagation of a transaction in the Bitcoin network in terms of propagation speed. The more monitoring nodes in the topology, the faster the transaction propagated throughout the network.

Size of the Bitcoin network

The average number of reachable Bitcoin nodes in the past year was about 17000 [12]. The question related to the number of nodes was whether the network should be simulated with a similar number of reachable nodes corresponding to the real Bitcoin network or with only a partial number. The differences between the results of the simulations with a large number of nodes (15000 and 10000) and partial number of nodes (1000, 2000) were in the average position values. In the simulation with a large number of nodes, the position values were higher, but this is due to the calculation of the position value, which is based on the number of nodes. In contrast, the calculation of the time stamp is not dependent on the number of nodes and the results are similar in both cases. Moreover, if the values from different sized simulations are recalculated proportionally, the results are similar. In addition, the number of outbound connections was exponentially scaled by the number of nodes to make the results as accurate as possible. However, the identification of source node was successful and the overall results were similar to the simulations with a small number of nodes. For this reason, it was decided that there is no need to simulate the network with a large number of nodes for this simulation scenario, since there was no effect on the overall result.

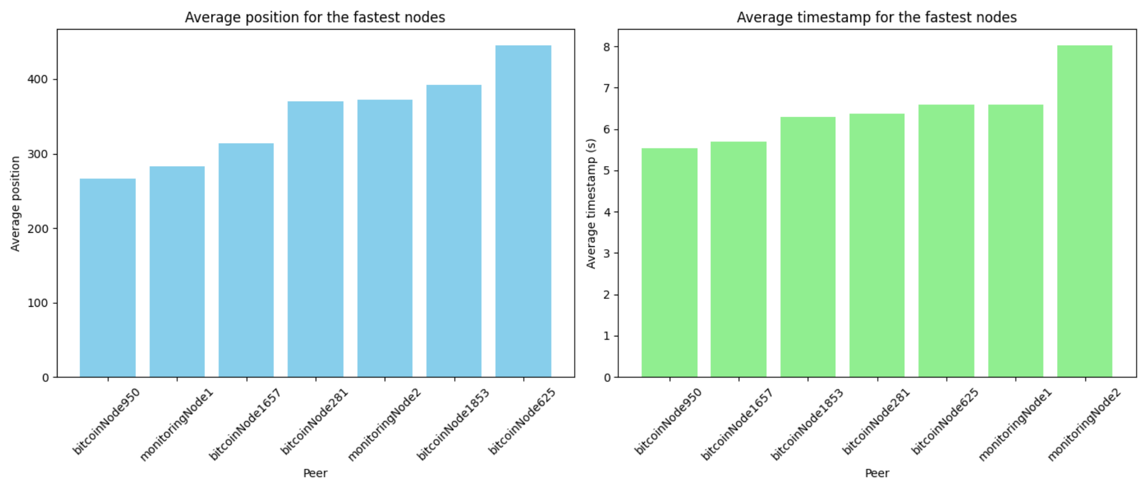


Figure 7.4: Results when there are multiple monitoring nodes in the network that propagate transactions when the **source node is behind NAT**.

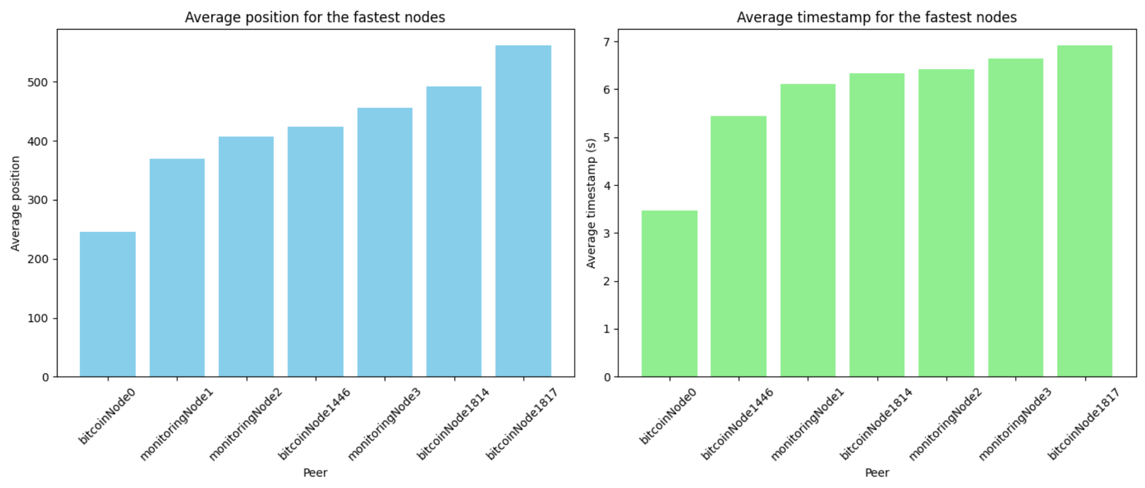


Figure 7.5: Results when there are multiple monitoring nodes in the network that propagate transactions when the **source node is not behind NAT**.

Chapter 8

Conclusion

A simulation model with the Bitcoin algorithm for propagating inventory messages was created in the OMNeT++ discrete-event simulator. In addition, the simulation model includes a monitoring node that connects to all reachable Bitcoin clients and collects a record of the transaction propagation. Then, this record was analyzed and the results described in the last chapter.

An automated script was developed to define the simulation parameters, set up the Bitcoin network topology, and run the whole simulation. The simulation covers situations where the source node of a transaction is behind NAT device or directly on the public network.

The main goal of the analysis was to identify the fastest nodes and the possible source node of the transaction. There are several reasons why a node might be the fastest. It could be the source node of the transaction, be located near the source, or have a different implementation that speeds up transaction forwarding. The analysis script was able to identify fastest nodes and determine the possible source node of the transaction, or atleast the node that was close to it, in case the source node is behind NAT. Another result obtained was that monitoring nodes can enable faster propagation of transactions and thus can affect the network in some way, if they propagate transactions as Bitcoin clients. It was also found that when scaling outbound connections correctly, there is no need to simulate a network with a real number of Bitcoin nodes (17 000), but only a partial one (2000).

The simulation model created using OMNeT++ can help understand how inventory messages propagate in the Bitcoin network. Simulations can be run under a graphical user interface such as Qtenv, where the entire transaction propagation flow can be seen and all events described. In addition, the model implements a monitoring node that can be used to further analyze the behavior of the Bitcoin network.

The implementation aspects of this thesis were presented in poster format at the Excel@FIT 2024 Student Conference[1]. In the future, the work could be improved by implementing Bitcoin getdata and tx messages, further analysis, and adding more options for simulation configuration and Bitcoin network.

In conclusion, this thesis aimed to summarize the theoretical side of blockchain technology and Bitcoin cryptocurrency while providing practical examples for a better understanding of the technologies or processes described. In the implementation part, a highly simplified Bitcoin client was developed, validated, and analyzed. Bitcoin client can create,

[1]https://excel.fit.vutbr.cz/

receive, and send messages representing Bitcoin transactions and operate according to the Bitcoin Core propagation algorithm.

Bibliography

[1] *Bitcoin - Open source P2P money* [online]. [cit. 2023-09-27]. Available at: https://bitcoin.org/.

[2] *Bitcoin Wiki* [online]. [cit. 2023-09-27]. Available at: https://en.bitcoin.it/wiki/Main_Page.

[3] Andresen, G. *Pay to Script Hash* [online]. 2012 [cit. 2023-12-09]. Available at: https://en.bitcoin.it/wiki/BIP_0016.

[4] Back, A. *Hashcash* [online]. [cit. 2023-10-22]. Available at: http://www.hashcash.org/.

[5] Bitcoin Core Repository. *Github* [online]. 2009, 2024 [cit. 2024-01-05]. Available at: https://github.com/bitcoin/bitcoin.

[6] Bitcoin Core Repository. *Reduce Traffic* [online]. 2015, 2023 [cit. 2024-01-05]. Available at: https://github.com/bitcoin/bitcoin/blob/master/doc/reduce-traffic.md.

[7] Bitcoin Project. Block Chain. *Bitcoindeveloper* [online]. [cit. 2023-12-20]. Available at: https://developer.bitcoin.org/reference/block_chain.html.

[8] Bitcoin Project. Transactions. *Bitcoindeveloper* [online]. [cit. 2023-12-20]. Available at: https://developer.bitcoin.org/reference/transactions.html.

[9] Bitcoin Project. P2P Network. *Bitcoindeveloper* [online]. [cit. 2023-12-20]. Available at: https://developer.bitcoin.org/devguide/p2p_network.html.

[10] Bitcoin Project. P2P Network. *Bitcoindeveloper* [online]. [cit. 2024-04-18]. Available at: https://developer.bitcoin.org/reference/p2p_networking.html#tx.

[11] Bitcoin Project. *Bitcoindeveloper* [online]. 2009, 2020 [cit. 2023-09-12]. Available at: https://developer.bitcoin.org/devguide/index.html.

[12] BITNODES. *Bitcoin Network 1 Year Chart* [online]. 2024 [cit. 2024-03-18]. Available at: https://bitnodes.io/dashboard/1y/.

[13] CoinDesk. *Understanding the DAO attack* [online]. 2016 [cit. 2023-11-22]. Available at: https://www.coindesk.com/learn/understanding-the-dao-attack/.

[14] Dashjr, L. *BIP process, revised* [online]. 2016 [cit. 2024-01-05]. Available at: https://en.bitcoin.it/wiki/BIP_0002.

[15] Friedenbach, M. *Relative lock-time using consensus-enforced sequence numbers* [online]. BtcDrak, Nicolas Dorier, kinoshitajona. 2015 [cit. 2023-12-18]. Available at: https://en.bitcoin.it/wiki/BIP_0068.

[16] Harding, D. A. *Opt-in Full Replace-by-Fee Signaling* [online]. Peter Todd. 2015 [cit. 2023-12-20]. Available at: https://en.bitcoin.it/wiki/BIP_0125.

[17] Hyperledger Foundation. *White Papers* [online]. [cit. 2023-11-27]. Available at: https://www.hyperledger.org/learn/white-papers.

[18] Jafari Navimipour, N. and Sharifi Milani, F. A comprehensive study of the resource discovery techniques in Peer-to-Peer networks. *Peer-to-Peer Networking and Applications*. May 2015, vol. 8, no. 3, p. 474–492. DOI: 10.1007/s12083-014-0271-5. ISSN 1936-6450. Available at: https://doi.org/10.1007/s12083-014-0271-5.

[19] Laurence, T. *Blockchain For Dummies*. 3rd ed. John Wiley & Sons, Inc., 2023. ISBN 978-1-394-15966-6.

[20] Lombrozo, E. *Segregated Witness (Consensus layer)* [online]. Johnson Lau, Pieter Wuille. 2015 [cit. 2023-12-24]. Available at: https://en.bitcoin.it/wiki/BIP_0141.

[21] Mei, L. *Blockchain, Bitcoin, and the Digital Economy*. Mercury Learning and Information, 2022. ISBN 978-1-68392-835-5.

[22] Nakamoto, S. *Bitcoin: A peer-to-peer electronic cash system* [online]. 2008 [cit. 2023-11-28]. Available at: https://nakamotoinstitute.org/bitcoin/.

[23] Rosenbaum, K. *Grokking Bitcoin*. Manning Publications, 2019. ISBN 978-1617294648.

[24] Schollmeier, R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: *Proceedings First International Conference on Peer-to-Peer Computing*. IEEE, 2001, p. 101–102. DOI: 10.1109/P2P.2001.990434.

[25] Shaikh, S. *Building Decentralized Blockchain Applications: Learn How to Use Blockchain as the Foundation for Next-Gen Apps*. BPB Publications, 2021. ISBN 978-93-89898-620.

[26] Smith, C. *NETWORKING LAYER* [online]. 29. march 2022. Revised 7.4.2023 [cit. 2023-12-26]. Available at: https://ethereum.org/en/developers/docs/networking-layer/#execution-layer.

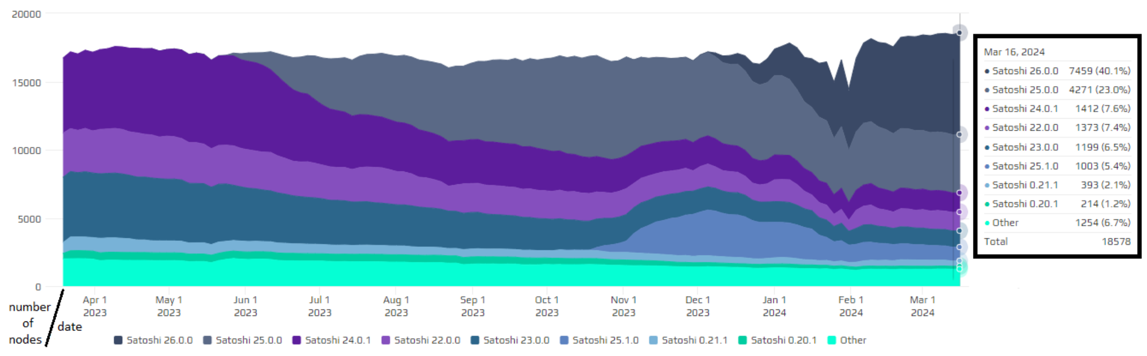
[27] Varga, A. *OMNeT++ Simulation Manual* [online]. 6th ed. OpenSim Ltd. OMNeT++ Community, 2023. Available at: https://doc.omnetpp.org/omnetpp/SimulationManual.pdf.

[28] Walker, G. *Learn me a bitcoin* [online]. 2018. 2023-10-09 [cit. 2023-12-09]. Available at: https://learnmeabitcoin.com/technical/script.

[29] Wuille, P. *Taproot: SegWit version 1 spending rules* [online]. Jonas Nick, Anthony Towns. 2020 [cit. 2023-12-20]. Available at: https://en.bitcoin.it/wiki/BIP_0341.

Appendix A

Distribution of reachable Bitcoin nodes across leading user agents.



Mar 16, 2024	
● Satoshi 26.0.0	7459 (40.1%)
● Satoshi 25.0.0	4271 (23.0%)
● Satoshi 24.0.1	1412 (7.6%)
● Satoshi 22.0.0	1373 (7.4%)
● Satoshi 23.0.0	1199 (6.5%)
● Satoshi 25.1.0	1003 (5.4%)
● Satoshi 0.21.1	393 (2.1%)
● Satoshi 0.20.1	214 (1.2%)
● Other	1254 (6.7%)
Total	18578

Figure A.1: The chart shows the number of reachable nodes over the last year, along with the distribution of different versions of Bitcoin clients. Bitcoin Core clients are the most widely used type of Bitcoin client in the world. As of March 16, they made up more than 93 percent of the Bitcoin network [12].

Appendix B

Contents of Included Disk Media

File	**Description**
thesis.pdf	Copy of this thesis in PDF format.
btc_network.zip	Zip file containing the source codes for the implementation part.
thesis_latex.zip	Zip file containing the source code for the generated PDF.
README.md	Instructions on how to set up the environment and run the simulation.

Table B.1: Contents of the included disk media.