



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

CRYPTOCURRENCY NODE MONITORING

MONITORING UZLŮ KRYPTOMĚNOVÝCH SÍTÍ

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

ANDREJ ZAUJEC

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. VLADIMÍR VESELÝ, Ph.D.

BRNO 2020

Zadání bakalářské práce



Student: **Zaujec Andrej**
Program: Informační technologie
Název: **Monitoring uzlů kryptoměnových sítí**
Cryptocurrency Node Monitoring
Kategorie: Počítačové sítě

Zadání:

1. Nastudujte si teorii za nejdůležitějšími kryptoměnami (Bitcoin, Ethereum, Ripple, EOS, Stellar, Cardano) a seznamte se s principy provozu klientů jejich peer-to-peer sítí, zaměřte se přitom zejména na komunikační protokoly a RPC API klientů.
2. Identifikujte relevantní metadata související se stotožňováním zařízení, na kterých kryptoklienti běží a navrhnete platformu pro jejich dlouhodobý sběr.
3. Dle doporučení vedoucího implementujte platformu z bodu 2 a pokuste se o souvislejší sběr metadat nad vybranou kryptoměnovou sítí.
4. Otestujte vaše řešení: a) analyzujte velikost peer-to-peer sítě v čase; b) zaměřte se na zhodnocení škálovatelnosti monitoringu. Diskutujte možná rozšíření.

Literatura:

- Narayanan, A., Bonneau, J., Felten, E., Miller, A., & Goldfeder, S. (2016). *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press.
- Bitpay, *Guides - Bitcore*, [online] <https://bitcore.io/guides>, [2018-10-19].

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 včetně.

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **Veselý Vladimír, Ing., Ph.D.**

Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.

Datum zadání: 1. listopadu 2019

Datum odevzdání: 28. května 2020

Datum schválení: 16. října 2019

Abstract

The goal of this bachelor thesis is to monitor nodes in the Bitcoin peer-to-peer network and to estimate the size of the network in a given time. Monitoring of nodes includes gathering metadata about them and creating activity records about how long they are participating in the network. The proposed solution uses the Bitcoin Core client without any modification to obtain all information about the network and nodes. The implemented platform enables gathering metadata (protocol version, user agent, services, IP address, port number) about found nodes. It obtains information about which nodes are in the network for the given time as well. Created API exposes collected data from the platform. Docker containers encapsulate each component of the platform and enable simple deployment within a few minutes.

Abstrakt

Cielom tejto bakalárskej práce je monitorovanie uzlov v Bitcoin peer-to-peer sieti a odhadnutie veľkosti siete v danom čase. Monitorovanie uzlov zahŕňa zbieranie metadát o daných uzloch a tiež aj vytváranie záznamov činnosti o tom ako dlho boli jednotlivé uzly súčasťou siete. Navrhnuté riešenie využíva Bitcoin Core klienta bez ďalších modifikácií na zistenie všetkých informácií o sieti a uzloch. Implementovaná platforma umožňuje zbieranie metadát (verzia protokolu, verzia agenta, ponúkané služby, IP adresa, číslo portu) o nájdených uzloch. Taktiež je schopná získavať informácie o tom, ktoré uzly sa náchadzajú v sieti v danom čase. Vytvorené API ponúka zozbierané dáta z platformy. Každá komponenta platformy je zapúzdrená v Docker kontajneroch čo umožňuje jednoduché nasadenie celej platformy v priebehu niekoľkých minút.

Keywords

scanning, peer-to-peer, nodes, monitoring, cryptocurrencies.

Klíčové slová

prehľadávanie, peer-to-peer, uzly, monitorovanie, kryptomeny.

Reference

ZAUJEC, Andrej. *Cryptocurrency Node Monitoring*. Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

Rozšírený abstrakt

Cieľom tejto bakalárskej práce je monitorovanie uzlov v Bitcoin peer-to-peer sieti a odhadnutie veľkosti siete v danom čase. Kryptomeny sa v posledných časoch tešia nevídanej obľube. Inak to nie je ani v prípade Bitcoinu, ktorý má najväčší podiel na trhu kryptomien podľa Coinmarketcap¹.

Fungovanie Bitcoinu stojí na niekoľkých hlavných pilieroch. Jeden z týchto pilierov je peer-to-peer sieť tvorenými uzlami. Uzly využívajú špeciálny vytvorený protokol na aplikačnej úrovni pre účely komunikácie. Jeden z príkazov v tomto protokole umožňuje opýtať sa uzlu na adresy ďalších uzlov, s ktorými má aktívne spojenie. Opakovaním volaním tohoto príkazu je potencionálne možné preskúmať celú sieť a odhadnúť jej veľkosť v danom čase. V prvej časti práce je vysvetlená základná problematika a motivácia tejto práce. V druhej časti sú naštudované potrebné znalosti ohľadom fungovania kryptomien, rozdelenie jednotlivých typov uzlov a tiež aj podrobnejšie preskúmané protokoly vybraných kryptomien ako Bitcoin, Ethereum, Ripple a EOS. Bitcoin protokol je rozobraný detailnejšie, ako ostatné protokoly a taktiež je tam preskúmaná možnosť prehľadania celej siete pomocou využitia volania spomínaného príkazu.

V tretej časti práce je rozobraná analýza metadát, ktoré jednotlivé uzly o sebe navzájom vedia a taktiež aj využitie zvolených metadát na jednoznačnú identifikáciu uzlov. Taktiež v tejto časti práce je navrhnutá platforma, ktorá sa zameriava na vyhľadávanie nových uzlov, naväzovanie aktívnych spojení, získanie zvolených metadát a vytváranie záznamov o dĺžke zotrvania jednotlivých uzlov v sieti. Platforma využíva oficiálnu implementáciu² Bitcoin Core klienta bez ďalších úprav. Všetky informácie ohľadom uzlov a siete sú čerpané z Bitcoin Core klienta pomocou jednoduchých troch volaní aplikačného rozhrania spomínaného klienta.

V štvrtej časti práce je objasnená implementácia navrhnuitej platformy a tiež sú tam detailnejšie popísané jednotlivé komponenty a princípy ich fungovania v danej platforme. Hlavné komponenty sú implementované v Pythone. Na komunikáciu medzi jednotlivými komponentami je používaná Kafka zatiaľ, čo na monitorovanie celej platformy pomocou metrík slúži Prometheus v kombinácii s Grafanou. Grafana vizualizuje pozbierané metriky a vytváraná z nich interaktívne grafy.

Vyhodnotenie a validácia získaných výsledkov z týždenného testovania platformy je v piatej časti práce. Jeden z výsledkov testovania dokazuje, že použitá platforma s Bitcoinovým klientom je konkurencieschopná oproti riešeniu platformy Bitnodes³, ktoré využíva len špecifické volania na úrovni Bitcoinového protokolu. Narozdiel od použitia Bitcoinového klienta a volania jeho aplikačného rozhrania, je riešenie od Bitnodes závislé na aktuálnej verzii protokolu a pri vydaní novej verzie protokolu sa stáva nekompatibilné pričom aplikačné rozhranie použitého klienta sa mení len zriedka.

¹<https://coinmarketcap.com/>

²<https://github.com/bitcoin/bitcoin>

³<https://bitnodes.io/>

Cryptocurrency Node Monitoring

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Vladimíra Veselého Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....

Andrej Zaujec

May 28, 2020

Acknowledgements

I would like to thank my supervisor Mr. Ing. Vladimír Veselý, Ph.D., for his endless support and patient guidance. As a sign of gratitude, here is a recipe for my favourite pancakes.

Put 100g plain flour, two large eggs, 300ml milk, 1 tbsp sunflower oil, and a pinch of salt into a bowl, then whisk to a smooth batter. Set aside for 30 minutes to rest. Set a frying pan over medium heat and carefully wipe it with some oiled kitchen paper. When hot, cook the pancakes for 1 min on each side until golden. Once cold, serve the pancakes with curd mixed with strawberry jam.

Contents

1	Introduction	3
2	Theory	4
2.1	Key Concepts	4
2.1.1	Blockchain	4
2.1.2	Distributed Ledger and Consensus	5
2.1.3	Proof of Work	5
2.1.4	Proof of Stake	6
2.1.5	Summary of Concepts	6
2.2	Nodes and Clients	6
2.2.1	Types of Nodes	6
2.2.2	Clients	7
2.3	Cryptocurrencies	8
2.3.1	Bitcoin	8
2.3.2	Ethereum	8
2.3.3	Ripple	9
2.3.4	EOS	9
2.4	Communication Protocol	9
2.5	Bitcoin Protocol	10
2.6	Other Cryptocurrencies Protocols	16
2.7	Bitcoin Core API	17
3	Analysis and Design	21
3.1	Analysis of Metadata	21
3.2	Platform Design	21
3.2.1	Logical Scheme	22
3.2.2	Database Model	23
3.3	Used technologies	24
4	Implementation	26
4.1	Wrappers	26
4.2	Database	27
4.3	Modules	29
4.3.1	Address_Puller	30
4.3.2	Address_Publisher	30
4.3.3	Node_Watcher	30
4.3.4	API	31
4.4	Metrics	32

5	Testing	33
5.1	Bitcoin Client Behaviour	33
5.2	Bitcoin Client Requirements	34
5.3	Summary of obtained data	35
5.4	Discovery of my node	37
5.5	Validating results	38
6	Conclusion	41
	Bibliography	42
A	Figures	44
B	CD contents	48

Chapter 1

Introduction

Cryptocurrencies are ubiquitous in the modern world. Nowadays, cryptocurrencies are seen more and more as a possible alternative to standard currencies. Even in some of the known chain stores is possible to pay with cryptocurrencies¹. Cryptocurrencies like Bitcoin, Litecoin, Ethereum, and Ripple are based on few foundations. Peer-to-peer (P2P) network and communication within this network are one of the foundations. Participants in the P2P network are called nodes. Nodes are accepting new connections, sending data to other nodes with which they have an active connection. Data exchange between nodes depends on the compliance of communication protocol.

Given protocol is determining everything, from begin of communication through connecting to nodes and sending data. In the case of Bitcoin, the protocol enables to ask node about his active connections. Thanks to this feature of the protocol, the adjusted node can crawl the whole P2P network and find most of the active participants. The network participants do not have an obligation to communicate and stay active the entire time. Because of this fact, it can be hard to predict network size and active nodes in the given time.

This bachelor thesis aims to crawl the Bitcoin P2P cryptocurrency network and to monitor found nodes. My primary motivation is to estimate network size by finding all nodes at the given time and gather metadata about discovered nodes. Another motivation for this thesis is to contribute with results to TARZAN² project. Thesis has following structure. Theoretical prerequisites are described in Chapter 2. Chapter 3 contains the analysis of metadata and the design of the platform. Chapter 4 describes the more in-depth details of the implemented platform. Chapter 5 provides information about the evaluation and validation of obtained data by the implemented platform.

¹<https://www.lifewire.com/big-sites-that-accept-bitcoin-payments-3485965>

²<https://www.fit.vut.cz/research/project/1063/.cs>

Chapter 2

Theory

The focus of this chapter is to explain theoretical prerequisites. The chapter starts with a brief description of several key mechanisms behind cryptocurrencies, and what are nodes in this context with possible types of nodes.

Then, an explanation of cryptocurrency clients and remote procedure call (RPC) application programming interface (API) follows. Furthermore, there are mentioned several cryptocurrencies with briefly described ideas behind them and with examples of their clients. Next, the chapter discusses the basics of protocol and its different versions.

The chapter ends with the description of chosen messages from the protocol and Bitcoin client API.

2.1 Key Concepts

There are several key mechanisms that the majority of cryptocurrencies have included in them. The common mechanism of these different cryptocurrency systems is the public ledger known as *blockchain* that is shared between network participants and the use of native tokens as a way to incentives participants for running the network in the absence of a central authority[8].

Other mechanisms are Proof of Work (PoW) and Proof of Stake (PoS). Both of these are variations of the consensus mechanism. A brief explanation of the mentioned consensus mechanisms will follow.

2.1.1 Blockchain

The transactions are at the lowest level of the hierarchy. These are collected and encapsulated into larger chunks of information called blocks. Blocks are linked together and create a chain called a *blockchain*. The next block always points to the previous block by holding a reference to it. This reference is a cryptographic hash of the previous block. Hashes secure the irreversibility of blocks, so there cannot be any further changes in already linked blocks. This applies to changes in the transactions as well, as blocks include them. Attempt to change block will end up in changed hash, so the references will not match.

The *blockchain* first block named *genesis block* is hard-coded into cryptocurrency software. Distance between any given block and *genesis block* is called block height. The lastly added block to the chain is a *head block*. The distance of the *genesis block* and *head block* determines the length of the whole chain.

Blocks have to meet the set of rules in order to be valid. This set of rules can vary, and users see these rules as one of the key parameters for the given cryptocurrency. Many cryptocurrencies are founded on the concept of a slightly different set of rules that are more comfortable for users than in other cryptocurrencies.

2.1.2 Distributed Ledger and Consensus

If *blockchain* was used in a centralized system that would lead to one central authority. This authority would have to determine which blocks should be added and how the *blockchain* current state should look.

On the other hand, when *blockchain* is distributed between several or more nodes without a single authority. The majority of participants in this distributed system has to step in and decide which block should be added. This majority of participants have to come for voting consensus. Distributed *blockchain* is falling into the category of distributed *ledgers*. The distributed ledger sits on multiple nodes at a given time. The nodes are communicating through the P2P network.

Each copy of the *ledger* lives independently. In such a dynamically changing status of the *ledger*, these publicly shared *ledgers* need a mechanism to ensure that all the transactions occurring on the network are genuine, and all participants agree on a consensus. Key features of this mechanism should be efficiency, fairness, security[4]. Synchronization of these independent copies is happening during voting in the chosen consensus algorithm.

Once a consensus has been determined, all the other nodes update themselves with the new, correct copy of the ledger[12]. In the context of cryptocurrencies, there are different kinds of consensus PoW and PoS mechanisms will follow.

2.1.3 Proof of Work

The Pow consensus mechanism is the widest deployed consensus mechanism in existing *blockchains*[5]. Participants are required to prove that work done by them qualifies them the newly created block that will append to the blockchain. The participants have to solve a computational puzzle in the case of Bitcoin.

The solution lies in finding the right *nonce*, so the hash of the block, including found *nonce*, is lower than the current target value. The *nonce* is a random integer that should be used only once. The provided solution is difficult for computing, but easy to verify. The right combination has to be just guessed, so it is a trial-error process.

Mining involves computing hash of the block that consists of unconfirmed transactions and *nonce*, during the mining process, only the *nonce* changes. In the case of a competitive scenario in cryptocurrencies, the mining process is heavy on computational resources because of a small-time limit due to other miners trying their best.

There is a reward awaiting miner for each mined block that ends up in the blockchain. This reward is the biggest motivation for miners. In pursuit of finding the right *nonce*, small miners are used to connecting to mining pools to increase their chances. Therefore, the reward for finding the right *nonce* is split equally per provided computational power for a given pool.

Security of the PoW mechanism is based on a condition that no person can gather more than 50% of current computational power in the system. If the condition is no longer valid, then such a person will easily control the adding of new blocks.

2.1.4 Proof of Stake

This mechanism is another possible variant for consensus. The PoS is designed to prolong the lifetime of cryptocurrency when many miners do not back the cryptocurrency.

One of the PoW problems is computational power, which comes in hand with high energy consumption. This implies that cryptocurrency overtime is only sustainable by providing more and more energy to it throughout its whole life.

Mining will be harder for cryptocurrencies, which are backed by many miners bypassing the time. Thus, it will not be that attractive for new miners, and cryptocurrency computational power may decrease. This can result in more attempts for 51% attack and controlling the new block addition.

The PoS is designed to save cryptocurrency from the stated scenario by giving mining power to users based on their coins' holdings. So the power of miner is reflected by his current stake of ownership.

For example, Bob, who holds 8% of some coins of given cryptocurrency, can only mine 8% of blocks for given cryptocurrency. To make a successful 51% attack, in this case, Bob would have to hold at least 51% of all coins for given cryptocurrency.

This version of consensus may raise questions of rather holding coins than spending it, which is not good at the end as well.

2.1.5 Summary of Concepts

To summarise, *blockchain* is an open, distributed ledger that can record transactions between two or more parties efficiently and in a verifiable and permanent way[9]. Adding new blocks of transactions into *blockchain* is given into the hands of the consensus mechanism.

There are many types of consensus mechanisms. The most used are PoW or PoS. Distributed *blockchain* coupled with the consensus mechanism, is one of the building stones used by many cryptocurrencies.

The majority of cryptocurrencies are largely clones of Bitcoin or other cryptocurrencies and simply feature different parameter values (e.g., different block time, currency supply, and issuance scheme)[8]. Many cryptocurrencies that came after Bitcoin show little to no improvements or innovations; therefore, these are referred to as *altcoins*.

2.2 Nodes and Clients

The network of nodes is what makes it possible for cryptocurrency to work. Therefore, nodes are responsible for acting as communication points that may perform different functions. Any device that connects to the network may be considered as a node in the sense that they are capable of communicating with each other.

Nodes are also able to transmit data between them. Data can contain various information about transactions, blocks, or other nodes. All this information is transmitted by different messages in the protocol.

2.2.1 Types of Nodes

Each node defines its type according to its particular functions. Two different types of nodes are described next.

Full Nodes

Full nodes are the ones that support and provide security to cryptocurrency, and they are indispensable to the network. These nodes may also be referred to as fully validating nodes as they engage in verifying transactions and blocks against the system consensus rules[1]. Also, full nodes can relay new transactions and blocks to the blockchain.

Full nodes download every block of transactions and check them against cryptocurrency consensus rules. The full node is not required to have a full copy of the blockchain itself, but it is recommended. The smaller size of the blockchain copy, the greater will be the danger of possible failure of the given node. The node with the full copy of blockchain can only be sure about the history of blocks. Full nodes form the backbone of the cryptocurrency by keeping a copy of the whole blockchain itself.

Most full nodes also serve lightweight clients by transmitting their transactions to the network and notifying them when a transaction affects their wallet. Unless enough nodes perform this function, clients will not be able to connect through the P2P network. In that case, the clients will have to use centralized third-party services instead.

Lightweight Nodes

The lightweight node or light node does not download the complete blockchain. Instead, it downloads the block headers only to validate the authenticity of the transactions[3]. Thus, lightweight nodes do not require as much space as full nodes and are easier to maintain and run for the users.

For the user to verify his transactions, lightweight nodes use a method called Simplified payment verification (SPV). The user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he is convinced he has the longest chain and the transaction to the block it is timestamped in[13].

Full nodes serve information about the cryptocurrency network to the lightweight nodes. Full nodes are allowing lightweight nodes to propagate their transactions to the cryptocurrency network as well. This means lightweight nodes can not function without full nodes and are entirely dependent on them.

2.2.2 Clients

Each node runs a specialized software called a client that accesses a service made available by a server. In terms of cryptocurrency, the server is one node in the P2P network, and service is representing the implemented protocol described by given cryptocurrency. The node is made from the device by installing the cryptocurrency client software and exposing it to the internet. Client software usually comes in two parts.

- Daemon — core of cryptocurrency program itself
- Graphical or Command-line interface — utility for communication with core program through API

The second part does not always have to be included as its primary reason is to make communication between *daemon* and user easier by implementing API calls as commands or buttons.

The *daemon* will try to find its peers with various techniques. Afterward, when the *daemon* is connected to some of its newly found peers, then the synchronization process

can start. In the final state, the *daemon* is holding a valid copy of blockchain itself and is helping to maintain the healthy system. The *daemon* is listening for new blocks and transactions, validating them, and sharing his current view of blockchain to other peers. These processes will be in-depth, described later in the thesis. The *Daemon* provides many services at the same time.

A Hypertext Transfer Protocol (HTTP) server that serves as a form of API is one of the daemon services. This API allows user communication with the running cryptocurrency core program. Javascript Object Notation (JSON) RPC is the most often used kind of API in cryptocurrencies software, where JSON-RPC is RPC encoded in JSON. This is a form of client-server interaction (caller is client, an executor is a server), typically implemented via a request-response message-passing system[10].

Whenever the user installs the client, implicitly user will as well administrate the given client. This is the reason why the default scope of APIs is the localhost. The mentioned scope can be extended. The user should protect his admin role with credentials in that case.

Important client settings as Domain Name System (DNS) seeds of the potential nodes or maximum size of blockchain are written into the configuration file before the client starts up. During the start, the client will read the configuration file, parse all valid commands, and adjust his default behavior according to those commands. Calls through client API can manipulate the behavior of the running client.

2.3 Cryptocurrencies

Despite the significant number of cryptocurrencies nowadays, there are several that stand out because of finances invested in them and the number of active users. A brief description of selected cryptocurrencies with their clients follows.

2.3.1 Bitcoin

Bitcoin belongs to one of the first cryptocurrencies. Bitcoin was invented in 2008 by an unknown person or group of people using the name Satoshi Nakamoto and started in 2009 when its source code was released as open-source software[6]. Bitcoin laid down foundations of cryptocurrencies with new key concepts that are used in many cryptocurrencies that were introduced after him.

The official bitcoin client is Bitcoin Core programmed in the C++ language, also known as Satoshi client. This client reflects all aspects of the official client in his implementation and is used as the reference in terms of creating other clients. The client code can be download from official github repository¹, then compiled and ready to run. The client is ready to serve as the full node after the initial download of the whole *blockchain*.

The light version of this client can be achieved with a pruning option. This option sets the maximum size of the last n blocks in megabytes, which will be downloaded.

2.3.2 Ethereum

Ethereum is also known as a decentralized computing platform with its Turing-complete programming language, Solidity, which is used in *smart contracts*. Ethereum uses the

¹<https://github.com/bitcoin/bitcoin>

blockchain scripts, known as *smart contracts*. The *smart contracts* are executed programs on the custom run-time platform called Ethereum Virtual Machine (EVM).

With smart contracts, a new way of creating applications emerged. New applications sitting in blockchain are called *dapps* abbreviated from *distributed applications*. Ether is naming for a token.

In comparison with Bitcoin, Ethereum has three official clients that differ in the implementation language. Ethereum clients are with official repositories:

- Aleth — C++ client²
- Geth — Go client³
- Trinity — Python client⁴

Mentioned clients are used as a reference for official clients, and third-party software can be derived from them by forking official repositories or copy code parts.

2.3.3 Ripple

Ripple has inherited a few characteristic traits from Bitcoin. However, it is not a consensus-based cryptocurrency but rather consensus-oriented. It is using the private global *ledger* instead of the public *blockchain*. This means, only nodes from the chosen group are allowed to write to this ledger.

Ripple, as well, proved itself in faster transaction processing and better scalability than the Ethereum and Bitcoin[15]. All Ripple tokens exist now, and other ones cannot be mined or minted. Instead, the Ripple tokens are destroyed with every transaction and serve as proof of destruction to protect the system from malicious attacks.

The official Ripple client is called *rippled*, which is implemented in the C++ language and can be download from official repository⁵.

2.3.4 EOS

EOS is in many ways similar to Ethereum as it aims to develop *dapps* in the blockchain. EOS is trying to improve some of the Ethereum drawbacks as *blockchain* scalability and transaction fees. EOS replaced the pay-per-transaction model, which is used in the Ethereum by PoS. Providing this change, developing and maintaining *dapps* is less bounded by transaction charges and can support many scaled *dapps*.

The official client is implemented in the C++ language, and its source code can be download from the github repository⁶.

2.4 Communication Protocol

Cryptocurrencies like Bitcoin and Ethereum have their custom protocol built on application level in the TCP/IP stack, whereas Ripple and EOS use WebSocket or HTTP with JSON-RPC. Communication between nodes uses these protocols. The custom protocol defines the rules, syntax, and semantics for communicators to follow.

²<https://github.com/ethereum/aleth>

³<https://github.com/ethereum/go-ethereum>

⁴<https://github.com/ethereum/trinity>

⁵<https://github.com/ripple/rippled>

⁶<https://github.com/EOSIO/eos>

Cryptocurrencies have multiple networks, where one network is main, and others serve for testing. Testing networks can differ from the main by using different consensus algorithms and developing another independent *blockchain*. Other testing networks may mirror the main blockchain and serve it for testing purposes.

2.5 Bitcoin Protocol

The main network in the Bitcoin network is known as Mainnet. There are two other versions of the Bitcoin network whose sole purpose is testing. These networks are Testnet and Regtest.

Testnet coins are separate and distinct from actual bitcoins and are never supposed to have any value. This allows application developers or bitcoin testers to experiment without using real bitcoins or worrying about breaking the main blockchain.

Regtest is the network for a local testing environment in which developers can almost instantly generate blocks on demand for testing events, and can create own coins with no real-world value. Bitcoin protocol uses port numbers and constant string in all messages to determine in which network the communication is happening. These constant strings and port numbers for mentioned networks are in Table 2.1.

All communication in the Bitcoin protocol is happening through the messages. The message consists of the header and optional payload. The message header always has four fields. Almost all integers are encoded in little-endian. Only IP addresses and port numbers are encoded in big-endian. All messages in the network protocol use the header format shown in Table 2.2. If the payload is empty, as in *verack* and *getaddr* messages, the checksum will always be 0x5df6e0e2. This checksum is computed from (SHA256(SHA256(<empty string>))).

Network	Default Port	Start String
Mainnet	8333	0xf9beb4d9
Testnet	18333	0x0b110907
Regtest	18444	0xfabfb5da

Table 2.1: Different Bitcoin networks

Field Size	Description	Data type	Comments
4	magic	uint32_t	Magic value indicating message origin network, and used to seek next message when stream state is unknown
12	command	char[12]	ASCII string identifying the packet content, NULL padded
4	length	uint32_t	Length of payload in number of bytes
4	checksum	uint32_t	First 4 bytes of sha256(sha256(payload))

Table 2.2: Message header

The Bitcoin protocol primary intention is to share information about transactions or blocks. Following Figure 2.1 shows messages that request or response with that kind of information.

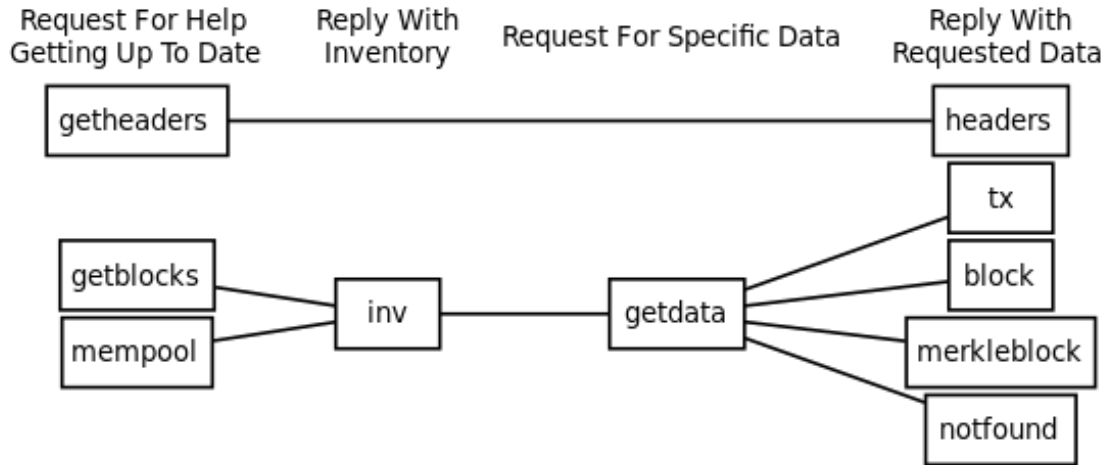


Figure 2.1: Data messages

Most used messages are enumerated and summarized in the list below. Messages, which are not included in the given list, are either deprecated or described later in the thesis.

- ping, pong – These messages are swapped between nodes in order to check if the connected peer is still active.
- version, verack – This message pair is used at the beginning of the connection. Firstly, *version* is sent from the node initiating a connection, and then response message with *verack* payload follows from the node, which accepts the connection.
- inv – The inventory is represented as *inv* message. Inventories serve as unique identifiers for information and are used by many data messages for blocks and transactions. Each inventory consists of the data type identifier and the hash of that object that serves as a unique identification.
- getheaders – Similarly, *getheaders* message requests for same information as *getblocks* but response is stored into *headers* message. Difference between responses is in maximum number of entries, while into *inv* can fit 500 entries, *headers* can store up to 2000.
- getblocks – This message requests *inv* of block header hashes, which are higher than start hash and lower than stop hash. Both start hash and stop hash are provided in *getblocks*. Hashes that satisfy the given range are returned in *inv* as the response.
- getdata – *Getdata* queries node for specific data object. Information about selected data object were provided in *inv* before *getdata*. If specified data object was found, *tx*, *block* or *merkleblock* message can be returned as response. On the other hand, if data object was not found *notfound* message is returned.
- mempool – *Mempool* message requests the unconfirmed validated transactions from node. Response for *mempool* is the view of the node on relayed unconfirmed transactions returned in *inv*. This response is the source for new transactions to be included in a new block by the miners.

A Bloom filter is a bit-field in which bits are set based on feeding the data element to a set of different hash functions[7]. This filter is used mainly by the SPV clients for requesting only matching transactions and blocks from the full nodes, which includes the SPV clients' wallets. The Bloom filter messages are described in the next list.

- FilterAdd, FilterClean, FilterLoad – These messages serve as settings for the Bloom filter of the receiving peer from the sending peer. The receiving peer will not relay transactions that do not meet announced settings to the peer that required filtered messages. The peer sets specific filter by sending *FilterAdd* or *FilterLoad*.
- FeeFilter – This message is also used by the node to announce its peers that it will not let any transactions below a specified fee into its memory pool.

Version message has to be exchanged always at the start of communication. Interesting payload attributes of this message with brief comments are in Table 2.3. Other attributes that are not in table are timestamp, addr_recv_services, addr_recv_IP_address, nonce, relay.

Name	Description
version	The highest version of protocol understood by the transmitting node.
services	The services supported by the transmitting node encoded as a bit-field.
user_agent	User agent with version to distinguish clients on network.
addr_trans IP address	The IPv6 address of the transmitting node in big endian byte order. IPv4 addresses can be provided as IPv4-mapped IPv6 addresses.
addr_trans port	The port number of the transmitting node in big endian byte order.
start_height	The height of the transmitting node's best block chain or best block header chain.

Table 2.3: Attributes of version message

Mentioned services identifiers from Table 2.3 are described in Table 2.4.

Peer Discovery

After client starts functioning, it has to find out another active node to be able to connect to P2P network.

Firstly, client checks its local database of the previous connections. If case local database is empty, client will try to get IP addresses by DNS lookup. Domain names used in lookups are called DNS seeds and originates directly from the source code of client. These DNS seeds are hard-coded into client source code and serve as a backup when everything else fails. Clients will always try to avoid querying DNS seeds unless they have to. If even DNS seeds fails and none of the known nodes are active, client last fall back will be a set of hard-coded IP of potential peers.

After the successful connection to one of IP addresses acquired from any of the previously mentioned ways, client uses another method for discovering new active peers. New method

Value	Name	Description
0x00	Unnamed	This node is not a full node. It may not be able to provide any data except for the transactions it originates.
0x01	NODE_NETWORK	This is a full node and can be asked for full blocks. It should implement all protocol features available.
0x04	NODE_BLOOM	This is a full node capable and willing to handle bloom-filtered connections.
0x0400	NODE_NETWORK_LIMITED	This is the same as NODE_NETWORK but the node has at least the last 288 blocks.

Table 2.4: Services identifiers

of peer discovery is by address rumoring, where connected peers gossip about other potential available peers. Peer will gossip only about peers with which peer has active connection. Peers sustains active connection by swapping ping, pong messages with each other.

Peer Connection

Connecting to a peer is done by sending the *version*. The remote peer responds with its own *version*. Then, both peers send a *verack* to each other to indicate the connection has been established, as can be seen in Figure A.3

Both nodes have to exchange *version* before any other communication happens. Until this exchange, no other messages will be accepted. The *version* provides key information about the transmitting node. The receiving node will send *verack* as a sign of acceptance. The payload of *version* is described below in Table 2.3.

The node can begin sending other messages after it receives *verack*. If the node wants to know about other active nodes, it will send the *getaddr* message. The *getaddr* requests an *addr* message from the receiving node, preferably one with lots of IP addresses of other nodes. The transmitting node can use those IP addresses to find out other available nodes rather than waiting for unsolicited *addr* to arrive over time.

In order to maintain a connection with the peer, the nodes by default will send the *ping* to peers before 30 minutes of inactivity. If 90 minutes pass without a *pong* being received by the peer, the client will assume that the connection has closed.

Downloading Copy of Blockchain

Before a full node can validate unconfirmed transactions and recently-mined blocks, it must download and validate all blocks from the *genesis block* to the current tip of the best *blockchain*[2]. This process of validation whole *blockchain* is called Initial Block Download or initial sync. It does not have to be done only once as word initial suggests.

This process is done regularly when the node has piled up a large number of blocks that were added to *blockchain* but the node did not download them. One of the cases is when the node had been offline for a longer time or there is a large number of blocks, which should be downloaded. The Bitcoin Core client does initial synchronization when the last block on the local *blockchain* has timestamp older than 24 hours or when the last block is

more than 144 blocks deeper than the newest best block. The newest best block is block which was added latest to the global *blockchain*.

The node at first start only contains one block in its local *blockchain*, this block is the *genesis block*. At this point, the node will choose his sync peer also known as sync node, to whom new node sends messages about blocks during the syncing process.

Bitcoin Core from version 0.10.0 uses a new initial synchronization method called headers-first. Instead of downloading the whole *blockchain* in order as in the previously used block-first approach, the headers-first method aims to firstly download the headers of *blockchain*, validate them as best as possible and then download blocks in parallel from multiple peers.

Initial sync starts with the node choosing the sync peer and sending him a *getheaders*. This message contains the header hash of the new node best local block. Stop hash is set to zeros in case the node wants as response maximum number of headers which can be 2000. The sync peer looks at the received header hash from *getheaders* and match local blocks with that header hash. The sync peer replies with all following blocks after that matched block until the block that matches stop hash from *getheaders*. In case stop hash is set to zeros, the sync peer replies with the maximum 2000 headers or less, it depends how many blocks follow matched header hash.

The new node will try to validate the received headers as much as possible by checking headers against the consensus rules, full validation can be only done with all transactions from that block. The new node can do 2 things in parallel after the partial validation of headers. It can download more headers or download blocks.

- Download more headers: The new node can repeat the same mentioned process with the sync peer. However, if the sync peer responds with less than 2000 headers on *getheaders* then the new node will send *getheaders* to all its active peers to see their views of the *blockchain*. By comparing multiple views on the *blockchain*, the initial sync node can quickly discover dishonest nodes and remove them from the active peers list.
- Download blocks: The new node will use information from download headers and request specific blocks with *getdata*. This request for the blocks is directed to several active peers of the new node. Thanks to this, the load is spread and does not depend on the upload of only one sync peer as in the process of downloading first headers.

The Bitcoin Core header-first mode is using the 1024 block moving window. If the whole block was not downloaded yet but Bitcoin Core is ready to validate it then Bitcoin Core will wait at least 2 seconds to the peer, which should send the block. If the block does not arrive after the waiting, Bitcoin Core client will disconnect from that peer and tries to connect to the another node. The summary of messages used in this mode is in Table 2.5.

Message	From ->To	Payload
getheaders	node ->Sync node	One or more header hashes
headers	Sync node ->node	Up to 2000 block headers
getdata	node ->active nodes	One or more block inventories computed from header hashes
block	active nodes ->node	One serialized block

Table 2.5: Header-first mode messages

Whenever sync is complete, the newly synced node is ready for listening and broadcasting the new blocks and the transactions created in the network.

Block Broadcasting

The nodes shares newly discovered blocks by the broadcasting. This can be done using one of the following methods.

- **Unsolicited Block Push:** The node shares the new block by block message to each of its peers. This can skip the standard relay method because none of the nodes peers knows nothing about the newly discovered block.
- **Standard Block Relay:** The node will send inventory with the newly discovered block encapsulated into *inv* to each of its peers. Block-first peers will request the new block using *getdata* while header-first peers will do this via the *getheaders* followed with the *getdata*.
- **Direct Headers Announcement:** The node can skip relay via *inv* and directly send *headers* with the new block header. The header-first peers will immediately validate the header and then request the block by *getdata*.

The node will reply to all the requests considering request type by sending *block*, *headers* or *tx*. The header-first peers can choose what they prefer between announcements using *headers* or rather using *inv*. The choice is made with special *sendheaders* message during connecting.

The default behavior in Bitcoin Core client is set to relay the new block via direct header announcement to all peers that requested it and use the standard relay to others.

Transaction Broadcasting

The transactions are relayed to peers using *inv*. *Getdata* should follow from peers who have an interest in the transaction. The transaction details are sent with *tx* message. Peers, which received transaction shares it to others in same way as described in case the transaction is valid.

Memory Pool

The unconfirmed transactions are essential for miners because these transactions can be used in the next new block. The unconfirmed transactions have to be relayed throughout the network to miners. This functionality is provided by full nodes that keep these transactions in their memory pools and move them around the network using transaction broadcasting.

The unconfirmed transactions are stored in non-permanent memory, so whenever the client is restarted, each unconfirmed stored transaction is deleted. Another way of deleting the unconfirmed transactions is the purging of memory for other transactions. The transaction is also removed from the pool in case it used in the newest added best block.

Mining the transaction represents adding the transaction to the block. The blocks, which were created but not used because the different block at the same height was used instead, are called stale blocks. The transactions used in stale blocks can be re-added to the pool. Never mined transactions tend to disappear from the network slowly. In fact, one study observed that 42% of all made transactions were not included in the *blockchain* after

one hour from their first appearance, 20% of the all made transactions were not included in the *blockchain* even after 30 days[14].

Misbehaving in Network

The nodes, which relays wrong information, results in misbehaving and increasing the ban score in the eyes of other nodes. The misbehaving causes are taking up a lot of bandwidth and computing resources. Ban score limit and ban duration in seconds are set in the configuration of the client. Each node maintains ban scores for its peers. Whenever one of the ban scores reaches the ban score limit for some peer, the peer will be banned for a period of ban duration. The default ban period is 24 hours.

Versions of Protocol

Many protocol versions have been developed throughout the years. Different versions are still used across nodes. Several notable versions of the Bitcoin protocol, with the most recent versions listed first are in Table 2.6.

Version	Release date	Major changes
70015	Jan 2017	New banning behavior for invalid compact blocks
70014	Aug 2016	Added sendcmpct, cmpctblock, getblocktxn, blocktxn messages
70013	Aug 2016	Added feefilter message, remove alert message
70012	Feb 2016	Added sendheaders message
70002	Mar 2014	Added reject message
70001	Feb 2013	Added notfound, filterload, filteradd, filterclear, merkleblock message

Table 2.6: Different versions of protocol

2.6 Other Cryptocurrencies Protocols

Ethereum Protocol

Ethereum uses network ID and the first block for the distinction of networks, these two attributes have to be the same at the beginning of nodes connection. Otherwise, the nodes will reject the connection to each other. Ethereum has more networks in comparison with the Bitcoin. A few examples of the Ethereum networks are in Table 2.7, and the actual list of all active networks with network IDs can be found on this website⁷. Each network is running on the port 30303.

Ethereum protocol is TCP/UDP based custom transport protocol build on messages similar to the Bitcoin protocol messages. Examples of the Ethereum protocol messages with name and the brief description are in Table 2.8.

Ripple

Ripple networks differ by domain names and ports. These attributes are set in the configuration file of the client. The default port used in networks is set on 51235. Domain

⁷<https://chainid.network/>

Network ID	Name	Network
1	Ethereum Mainnet	mainnet
4	Ethereum Testnet Rinkeby	rinkeby
42	Ethereum Testnet Kovan	kovan

Table 2.7: Examples of Ethereum networks

Message	Brief description
status	Inform peer of its state after established connection
newblockhashes	Inform about one or more block hashes which appeared in network
transactions	Inform peer about one or more transactions that it should know about
getblockheaders	Request peer for a information about block headers
blockheaders	List of block headers as response for getblockheaders message
getblockbodies	Request peer for a information about block bodies
blockbodies	List of block bodies as response for getblockbodies message
newblock	Inform peer about single block that is should know about
getnodedata	Request peer for a information containing state tree nodes
nodedata	Provide set of state tree nodes
getreceipt	Request peer for a information about receipts of block hashes
receipts	Provide set of receipts

Table 2.8: Ethereum protocol messages

names are DNS seeds for the client; these seeds will resolve in IP addresses of active peers. Domain names for Ripple networks are in Table 2.9.

Ripple nodes are using HTTP and WebSockets protocols on the application layer. Communication between nodes is happening through JSON-RPC requests. The same communication channel is used for communication between user and client. Method calls are divided into public and admin. Admin methods are available only if the user connects from a specified host and port, and these attributes are set at the configuration file. Method names with brief description are in Table 2.10

2.7 Bitcoin Core API

A description of the Bitcoin Core API calls will follow. Cryptocurrency clients are often using JSONRPC in order to communicate with the running daemon, as mentioned in Subsection 2.2.2. This holds for the Bitcoin Core client as well. The format of the Bitcoin Core

Domain name	Network type
s1.ripple.com	mainnet
s.altnet.rippletest.net	testnet
s.devnet.rippletest.net	devnet

Table 2.9: Ripple networks

Method name	Brief description
<i>Public methods</i>	
ledger	Request information about the public ledger
tx	Request information about the single transaction
submit	Send unconfirmed transaction to the network
subscribe	Request periodic notifications from peer
server_info	Request the server for a human-readable information about the server
ping	Test connection status and latency
<i>Private methods</i>	
connect	Force server to connect a specific peer
stop	Gracefully shut down the server
ledger_accept	Force the server to close current ledger and move to the next ledger number

Table 2.10: Ripple client methods

RPC is based on JSONRPC specification⁸. Valid HTTP request with JSONRPC is sent to API, processed by the client, and the response will follow. The request and response attributes with types, names and brief description are in Table 2.11 and in Table 2.12.

The **method** attribute is only required part of the RPC request whereas **params** attribute is not, as mentioned in Table 2.11. This is because many API methods do not require any parameters at all. Bitcoin Core client API does not expose the direct use of protocol messages. The client provides many API calls with the limited scope of use, instead. Some of these API calls encapsulates sending protocol messages. However, any

⁸https://www.jsonrpc.org/specification_v1

Name	Type	Required	Description
jsonrpc	number	False	Version of the RPC request.
id	string	False	String that will be returned with response.
method	string	True	The RPC method name.
params	array	False	An array containing parameters values for RPC method.

Table 2.11: JSON-RPC request attributes

Name	Type	Presence	Description
result	any	True	JSON response object
id	string	True	String that was provided with request.
error	object	True	The object containing description of occurred error, otherwise Null.
code	number	True	The error code number.
message	string	True	Text description of error or empty string

Table 2.12: JSON-RPC response attributes

abuse of these API calls is secured by the open-sourced implementation, which results in zero mistakes. In this thesis, basic API calls are used without any modification of the Bitcoin Core client code. Thus, there are a few API calls, which can be used to create new connections and monitor network. Other API calls can provide information about the current daemon knowledge or status of the daemon. Description of the several API calls with its names, parameters, and responses will follow next.

getpeerinfo

The *getpeerinfo* method returns information about connected peers as JSON objects in the array. Each object represents one connected peer. There are many attributes inside these objects. The most interesting ones for this thesis are in Listing 2.1. This API call does not require any parameters.

```
[
  {
    "addr":"host:port", // The IP address and port of the peer
    "subver": "/Satoshi:0.10.0/", // The string version
    "version": 70015, // The peer version
    "services":"00000000000000409", // The peer services
    "conntime": 1589906318, // The connection timestamp
    "relaytxes":true, // Peer asked for relaying transactions
    "lastsend": 1589906170, // The timestamp of the last send
    "lastrecv": 1589906133, // The timestamp of the last receive
    "bytessent": 392, // How many bytes were sent to peer
    "bytesrecv": 1046, // How many bytes per received by peer
    "timeoffset": 0, // The time offset in seconds
    "pingtime": 0.797432, // Average duration of ping round-trip time in seconds
    "minping": 0.207591, // Minimum round-trip time observed in seconds
    ...
  },
]
```

Listing 2.1: One object of *getpeerinfo* response

getnodeaddresses

The *getnodeaddresses* RPC method returns information about addresses that were seen by the node. These addresses can be used to discover new peers. Addresses are returned as objects in the array. One address object has several attributes, which are in Listing 2.2. This call accepts one optional parameter that represents how many addresses it should return. The maximum number of possible addresses returned in one call is 2500. The number of addresses sent by the client is set to around 23% of the total number of stored addresses without exceeding the threshold of 2500[11].

```
[
  {
    "time": 1589888960, //The timestamp when address was last seen
    "services": 1037, //Which services should be supported by node with this address
    "address": "2403:5800:c100:cf00:14bf:7b10:a84e:b45", //IPv4 or IPv6 address
    "port": 8333 //The port number
  },
  {
    "time": 1589193445,
    "services": 1037,
    "address": "77.191.93.43",
    "port": 8333
  }
  ...
]
```

Listing 2.2: Two objects of *getnodeaddresses* response

addnode

The main use of *addnode* is to either add the new peer or remove the early added peer from the client. The *addnode* RPC behavior depends on the provided parameters. It requires 2 parameters, first one is action, which can be *add*, *onetry*, *remove*. The second parameter is the address that will be added or removed. The *add* option secures that the client will try to keep initiating the connection with the provided address if no active connection exists. But, the *add* option can be activated with only eight possible addresses at a time. The client keeps an internal list of the addresses requested with the *add* option.

In contrast, the *onetry* option provides only one attempt to initiate the connection, but it can be used without limitation. This implies that if 2000 calls with the *onetry* option attempts turns out successful, the client will obtain 2000 new peers. The *remove* option is used for removing addresses that were requested with the *add* option. The requested address with the *remove* option will be disconnected if there is an active connection with such an address, and no other connection attempts are directed to that address. The *addnode* RPC returns JSON Null in the result attribute of the response.

Chapter 3

Analysis and Design

This chapter describes the analysis of collecting data and the platform design for monitoring nodes and collecting metadata about found nodes. The first section contains the metadata analysis, followed by the platform design in the second section. The third section focuses on the of the used technologies by the proposed platform.

3.1 Analysis of Metadata

Metadata can be described as data about other primary data. More information about primary data can bring up and show unseen relations. Metadata can also help with a better understanding of data sources.

In the case of this thesis, metadata of the found nodes can lead to more interesting notice about the used client software, which can result in the partial identification of the found nodes. In this context, the primary data are the IP addresses and the ports of found nodes, metadata are somewhat valuable data, which node implicitly told us during the regular communication.

From my point of view, interesting metadata are sent during *version* message. Nodes exchange this message during the initial connection, and nothing can be sent before the swap of the this message, as mentioned in Section 2.5. The Bitcoin Core client stores obtained metadata from the *version* message inside its internal database and expose this data via *getpeerinfo* RPC, as mentioned in Section 2.7.

The *version*, *services*, and *user_agent* are the interesting attributes of the *version* message. The *version* indicates the highest Bitcoin protocol version understood by the node. The *services* attribute represents the bit field of supported service by this node, these *services* with their bits mapping are in Table 2.4. The *user_agent* attribute is the string identifying the client software, recommended format of this string is /Name:Version/Name:Version/ according BIP14¹.

3.2 Platform Design

This section aims to describe the proposed platform with the component diagram and the database models.

¹<https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki>

3.2.1 Logical Scheme

The platform is created from six components. Every component is responsible for doing a specific job. Proposed component diagram of platform is in Figure 3.1. Rectangle with round edges represents components, and data flows between components are represented as oriented edges with arrows. Each component is briefly described then.

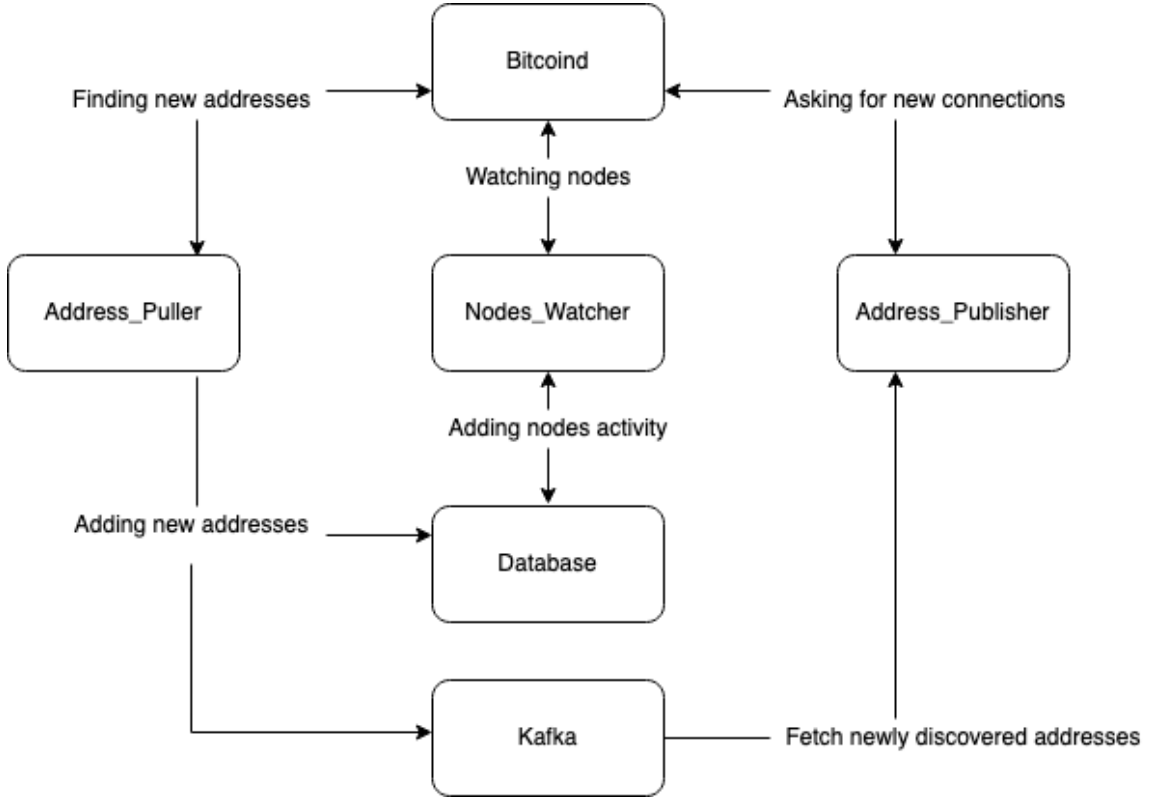


Figure 3.1: Component diagram

Bitcoin

As the name suggests, this component is the Bitcoin client, which other components will use for its functionality. *Bitcoin* component will not be implemented by me. Instead, the official Bitcoin client named Bitcoin Core will be used. Components communication with the *Bitcoin* component will be using JSON-RPC, as mentioned in Subsection 2.2.2.

Address_Puller

This component is responsible for pulling new addresses from the *Bitcoin* and comparing them with already obtained addresses. If some new addresses are present, the component will send it to the database and Kafka as well.

Nodes_Watcher

The aim of this component is to watch already connected nodes and to record nodes activity. Data about nodes activity and connection will be gathered from the *Bitcoin*. Activity records are created from the start time of activity and the end time of activity. The time

difference between these two periods is active node time in which node is considered active with adequately responding to the *ping* messages of the *Bitcoin*.

Address_Publisher

This component is asking Kafka for newly discovered addresses. These addresses are used for asking *Bitcoin* to try initiating new connections. One address represents one job for the worker inside this component. If the connection with node turns out to be successful, *Node_Watcher* component will take of monitoring and gathering metadata.

Database

This component represents a real Database, where all data gathered from components will be stored. The structure of data as schemes of tables or relations is described next in the thesis. *Address_Puller* and *Nodes_Watcher* are only two components that require this component for their proper work. .

Kafka

Kafka component contains a queue of tasks to process. In this context, one task represents one address to process. Address exchange between *Address_Puller* and *Address_Publisher* is done only via Kafka. This component will no be implemented by me.

3.2.2 Database Model

The schema of database entities and relations between them is in Figure 3.2 and entity description is provided below.

Node

Node table row represents one node and gathered metadata about the given node. Attributes in the table are *user_agent*, *highest_Protocol*, *services*, *active*. Node table is in relation one to many with *node_activity* table. The node can have many activities because of possible disconnecting and reconnecting of the node after some time.

Node_Activity

Node_Activity table has attributes *start_of_activity*, *end_of_activity*, *node_id*, which is a foreign key to the *node* table. The *start_of_activity* column represents the value when the active connection node was established. On the other hand, *end_of_activity* stands for the value when the node was no longer seen in the Bitcoin client active peers.

IP_Pool

The one row of *ip_pool* table is representing one found address, which consists of the IP address and port. Attributes of this table are *ip*, *port*, *inserted*, and *last_seen*. The *ip_pool* table is in relation one to one with *node* table. This relation is created for rows in *ip_pool* table, which was successfully used as node addresses. The primary key is created from a combination of the IP address and the port number.

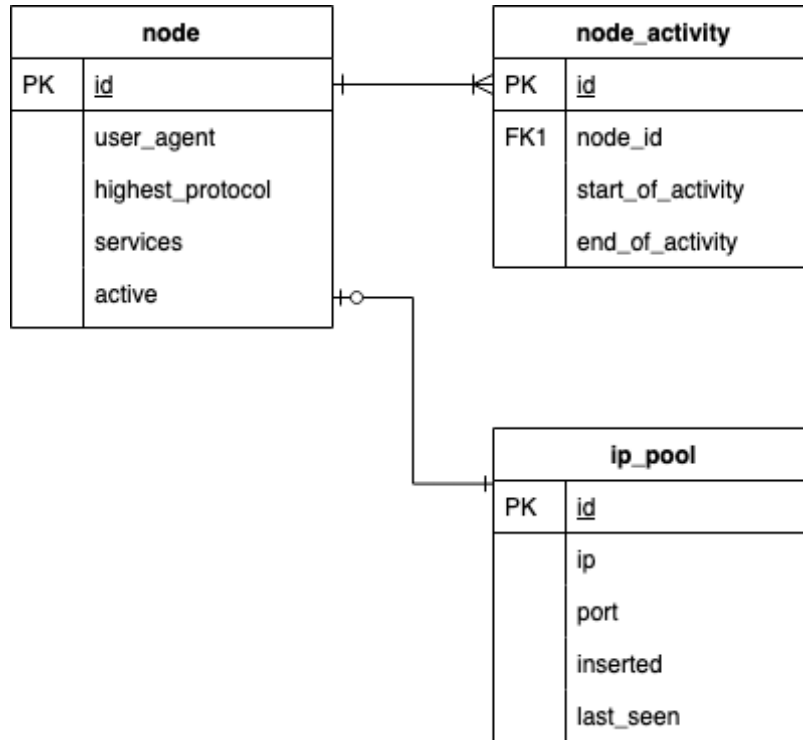


Figure 3.2: Entity-relationship diagram

3.3 Used technologies

Description of used technologies and their key features will follow in this section.

Python

Python is an interpreted, high-level programming language with a focus on cleaner syntax due to white spaces. White-space oriented syntactic rules and dynamic type checking allows quick prototyping and faster development. It matches well for various applications, which do not require precise management of the allocated resources. Suited applications can be ranged from network communication to system administration and many more. Python is vastly supported by a huge community of active developers. The community around Python creates and maintains many packages. Many packages are distributed under Open licenses and free to download.

PostgreSQL

The database system will handle a lot of queries from several services at a time. This load has to be handled correctly with minimum effort in minimum time. PostgreSQL was chosen as a tool for this job. PostgreSQL is an open-source database management system designed for handling a range of workloads with many concurrent users.

Kafka

Kafka aims to provide a high-throughput streaming platform. Data streaming is done via the producer and consumers principle. Streams are divided into topics. Internal data streams are called logs, which consist of messages. New data are represented as messages that append to logs. Messages are persistent and deleted after a period of time specified in the configuration. Kafka is internally managing several log offsets. Each offset reflects the current position of the consumer. Consumers are receiving messages from subscribed topics; meanwhile, producers are sending data to selected topics. One of the many key features is internal load-balancing through topic partitions. One topic can have many partitions, and the topic can only have as many consumers as partitions. This assures unique messages per consumer, so consumers do not have to read the same message twice or more.

Docker

Docker service provides isolated applications as packages called containers. Container contains a new specific environment ready for application run. One container often represents only one application. More robust applications are created from stacked containers. Docker solution is based on a single kernel. Thus, isolation is at the operating system level, which results in wasting fewer resources as opposed to the kernel level virtualization. Containers are communicating with each other through defined network adapters.

Prometheus

Prometheus is a real-time monitoring and alerting system. Metrics collecting is based on the HTTP pull model. Prometheus application regularly pulls plain text metrics by HTTP requests from exporters in defined intervals. The application can become an exporter by exposing HTTP endpoint with metrics. Every scraped agent has to be defined in the Prometheus scraping configuration. Prometheus is an excellent open-source solution in collecting and storing metrics. Prometheus also offers rich and flexible functional query language known as PromQL.

Grafana

Grafana is used in order to interactively and adequately visualize collected metrics into graphs. Grafana can be powered by various data sources. Prometheus belongs to one of these sources. Graphs are created over the collected Prometheus metrics. Graphs can be sorted into dashboards. The dashboard serves as a general overview of monitoring objects.

Chapter 4

Implementation

The goal of this chapter is to describe how the monitoring platform is implemented. The implemented platform is based on requesting Bitcoin Client via RPC. The client is periodically asked to show his active peers and to create new connections. Repetitive watching of active connections results in knowledge about the activity of other nodes. The implementation can be divided into these parts: Wrappers, Database, Modules, Metrics.

4.1 Wrappers

The implemented platform is doing communication with third-party services, which results in integrating third-party packages into code. I used object-oriented programming (OOP) pattern Wrapper in order to keep the code base clean of direct use of the third-party packages. Wrapper pattern recommends using classes for encapsulation of other interfaces. Wrapped classes add another layer of abstraction to the core logic. Thus, core logic protects itself from changes in third party code by using wrapped classes. In this case, wrapped interfaces are from packages **requests** and **Kafka-python**.

Requests package is used for doing HTTP communication. HTTP requests sent by platform are carrying JSON-RPC payloads. These calls are used to communicate with *Bitcoin*d as mentioned in Section 2.7. The public methods of *RequestWrapper* class are in Listing 4.1. Each one of these methods encapsulates execution of one RPC command. Used commands are next to method declarations.

```
class RequestWrapper:
    def fetch_addresses(self): //uses getnodeaddresses
        pass
    def post_new_peer(self, address): //uses addnode
        pass
    def fetch_active_peers(self): //uses getpeerinfo
        pass
```

Listing 4.1: The RequestWrapper methods

The flow of methods is almost the same. It only differs in the used RPC command. Firstly, the payload is serialized into JSON. Then, the request object with the serialized payload creates and is sent via the interface of **requests** package. Only *post_new_peer* method requires the parameter, which is the address of the potential node.

The **Kafka-Python** package is used for communication with running Kafka. What Kafka is and its purpose in this platform is explained in Section 3.1. I created the wrapped classes named *Producer*, *Consumer*. These wrapped classes are encapsulating direct use of *KafkaConsumer* and *KafkaProducer* classes from **Kafka-Python** package. The main part of the *Consumer* class is in Listing 4.2.

```
class Consumer:
    def __init__(self, topic=settings.DEFAULT_TOPIC, group_id=None):
        self.consumer = kafka.KafkaConsumer(
            topic, // The name of the topic to subscribe
            auto_offset_reset="earliest",
            enable_auto_commit=True,
            group_id=group_id,
            value_deserializer=lambda m: json.loads(m.decode("utf-8")),
        )

    def get_message(self):
        return next(self.consumer).value
```

Listing 4.2: The Consumer wrapper.

The *Consumer* class initialization creates *KafkaConsumer* with proper configuration. Provided attributes ensure that *Consumers* will automatically commit his current position in the reading topic. As well as, every *Consumer* uses the same *group_id* and value deserialization procedure. Thanks to the same group id, *Consumers* will have unique messages. If the *group_id* is not the same, *Consumers* will obtain the same messages. The *Consumer* class only exposes the *get_message* method. This method provides getting value directly from the new message.

```
class Producer:
    def __init__(self, topic=settings.DEFAULT_TOPIC):
        self.topic = topic
        self.producer = kafka.KafkaProducer(
            value_serializer=lambda m: json.dumps(m).encode("utf-8")
        )

    def post_message(self, message):
        self.producer.send(self.topic, message)
```

Listing 4.3: The Producer wrapper.

The *Producer* class initialization creates *KafkaProducer* with the same topic as *Consumer* in Listing 4.3. The value serialization method has revert order as in *Consumer*. This class exposes only *post_message* method used for posting new messages to Kafka.

4.2 Database

In order to fully understand core modules, work with the database has to be explained first. The schema and relations of the database are in Subsection 3.2.2. I decided to use

object-relational mapping (ORM) while working with the database instead of writing raw Structured Query Language (SQL) queries. Using ORM over raw queries has its pros and cons. One of the pros is that the ORM package will generate the SQL queries based on the work with created database models. On the other hand, the generated queries are often slower in execution than written queries for that specific task. I used *SQLAlchemy* as Python ORM packages. Working with database rows and columns in an object manner requires database models first to be created. An example of the Node database model implemented using the *SQLAlchemy* package is in Listing 4.4.

```
class Node(BASE_MODEL):
    __tablename__ = "node"
    id = Column(BigInteger, primary_key=True)
    user_agent = Column(String)
    active = Column(Boolean)
    highest_protocol = Column(String)
    services = Column(ARRAY(String))
    activities = relationship("NodeActivity")
    ip_pool = relationship("IP_Pool", uselist=False, back_populates="node")
```

Listing 4.4: The Node model class.

The implementation of the Node class reflects the Node entity from proposed ER diagram. The instantiated Column class object with specific data type class creates columns inside the model class, and both the Column and data type classes are imported from *SQLAlchemy*.

Every class that represents the database model has to inherit from BASE_MODEL class, which is an instance of declarative_base class. Thus, *SQLAlchemy* tracks every change of models into declarative_base class. This enables mapping objects onto database entities possible. Thanks to the tracking process, database migration can be generated as well by comparing two declarative_bases classes. I used Alembic for generating database migrations from SQLAlchemy declarative_base.

The two prerequisites for migration are the reference for currently used declarative_base in alembic/env.py and the database connection string in the alembic.ini file. The Alembic package installation process creates both of these mentioned files. If everything is set up correctly, then this series of commands in Listing 4.5 will generate the new database migration and apply its changes to the linked database.

```
alembic revision --autogenerate -m "Revision message"
alembic upgrade head
```

Listing 4.5: The database migration commands.

The platform communicates with the database only through transactions module nested under the database package. All database related code is nested under the database module. The example of transaction calls with names and their category based on the transaction behavior is summed up in the following dotted list. Transaction behavior is either getting data or saving new data.

- Save:

```
from app.database import transactions
```

```

transactions.new_ip(ip_attributes)
transactions.save_peer_activity_record(peer_attributes)
transactions.create_new_peer(peer_attributes)

```

- Get:

```

transactions.find_node(ip, port)
transactions.find_nodes(ip)
transactions.find_address(ip, port)

```

4.3 Modules

Firstly, the core logic divides into three main modules. These three core modules are *Address_Puller*, *Address_Publisher*, *Node_Watcher*. Every mentioned module can be executed separately and does not depend on the other modules. The main reason for this separation is to enable horizontal scaling of modules.

The brief description of each module is in Section 3.1. The configuration of modules via the settings module is described next. Then, how each module works is explained. This section ends with the description of the API module and the guide on how to start the platform.

Settings

The only thing that should vary in application for different deployments or instances is configuration. The values from the configuration file are loaded by the settings module. Then, the settings module contains loaded configuration as python variables and is used by main modules. The configuration is loaded from .env file, which is plain text file with key-value format. The .env file should reside in home directory of app. The example of .env file will follow. The modules that require specific key-value pair are written in bash comments format.

```

#every module requires these
HTTP_USER=user
HTTP_PASSWORD=password
DB_URL=connection_string

#address_puller
NODES=http://147.229.14.116:39999/bitcoin_rpc,

#address_publisher and node_watcher
NODES_TO_ACCEPT_CONNECTIONS=http://147.229.14.116:39999/bitcoin_rpc,
WORKERS=3

#address_publisher and address_puller
TOPIC_NAME=bitcoin_test_1
KAFKA=localhost:9092

```

In case when the user wants to pass multiple addresses into specific variables as `NODES`, the comma is used on the value string as the separator. Then, the variable contains the array from the comma divided sub-strings.

4.3.1 Address_Puller

The *address_puller* module is responsible for fetching addresses from the Bitcoin clients found in the `NODES` variable. If the address is new, it will be sent into the database and Kafka as well. The addresses are obtained by calling *fetch_addresses* method from *RequestWrapper* class. The RPC method behind this method call is *getnodeaddresses*. This method is repeated with every address that is set in `NODES` variable. Then, *address_puller* will set himself to sleep for 1 minute.

4.3.2 Address_Publisher

In the beginning, the *address_publisher* module will create several threads to fasten the address publishing process. The threads are called workers inside the implementation. The main thread creates a thread-safe queue object and *Consumer* object besides workers. Then, the main thread starts workers' processes and pass them the reference to the created queue. Meanwhile, workers are running. The main thread is periodically filling the queue with addressed gained from *Consumer* via *get_message* method calls.

The worker process takes the address of potential peer from the queue and sends it to the Bitcoin Client via *post_new_peer* method from *RequestWrapper*. In the low level, the RPC command *addnode* with first parameter *onetry* and second parameter obtained *address* is sent. The combination of *onetry* parameter with *addnode* method secures that the Bitcoin Client will give one try to establish the connection with the potential peer behind the provided address, as mentioned in Section 2.7. If a new connection is established, the Bitcoin client will have a new peer, which can be watched. This is very crucial for my thesis.

The number of created workers is set in `WORKERS` variable. The requests are sent to addresses in `NODES_TO_ACCEPT_CONNECTIONS`.

4.3.3 Node_Watcher

The *node_watcher* module job is to monitor all active peers and gather metadata about them. Firstly, *fetch_active_peers* from *RequestWrapper*. The response contains all active peers of the requested Bitcoin client as dictionaries in the array. One peer and data about him are represented as one dictionary object, as mentioned in Section 2.7 with example. Every peer from the request is transformed into the *Peer* object. The *Peer* object contains the IP address, port number, and interesting metadata about given active peer. The chosen metadata with attribute names and description are:

- **version:** highest known protocol by node
- **subver:** user agent
- **servicesnames:** list of supported services
- **conntime:** start of activity

More information about chosen metadata is mentioned in Section 3.1.

Created objects of *Peer* class are added to the *set of active peers*. This set represents all active peers found during the last active peers check. In Python, the set data structure contains objects with unique hashes only. The hashing method of the *Peer* class returns hash from concatenated IP address and port number. Thanks to this, only the first peer with the given IP address and the port number is watched. So no duplicates will ever occur in case of watching multiple nodes at once because of the hash collision.

When *node_watcher* adds the object of *Peer* class to set, it means that a new active peer was found. Then, *node_watcher* calls the node creation transaction with *Peer* object attributes and saves the newly found peer into to the database.

Added peer stays in *set of active peers* until that peer does miss in all responses from previous *fetch_active_peers* calls. This means the peer disconnected from all of the watched nodes. Record of activity is created with the database transaction and linked to the disconnected peer's node record.

One may raise a question that if this application is killed, all active peers will be forgotten and end up without any activity records. The *node_watcher* creates activity records for all currently active peers whenever it receives a kill signal in order to prevent loss of peer record from happening. The *node_watcher* module is doing checks of active peers every second.

4.3.4 API

The API module serves as a data provider between the database and the user. This way is collected data exposed in a more user-friendly and secure manner. There are three endpoints providing data about found nodes and addresses. The OpenAPI 3 standard describes the URL parameters and returned data from responses. Full OpenAPI 3 specification can be found on this link¹. Each response from endpoints is validated against defined data models to ensure compatibility with the described OpenAPI schemes. The description of endpoints, parameters, schemas are at the **docs** path when the API module is running. List with endpoint names and the brief description of what endpoint returns is in list below.

- **/node/{ip_address_with_port}** : This endpoint returns all gathered data about given node with list of all activities by the provided IP address and port number.
- **/nodes/{ip_address}** : This endpoint returns data about every node find behind the provided IP address. The data has same structure as **/node** endpoint response but it is listed in array.
- **/address/{ip_address_with_port}** : The address endpoint returns data from *ip_pool* table about the provided IP address and port. This will be used to find out if the platform sees given address.

The API returns the response with Not Found HTTP status code in case there is no data in the database about provided parameters.

Deployment of modules

Each core module can run as an independent docker container. These commands start the whole monitoring platform with metrics. Correctly filled *.env* file has to exists first as well as the database with up-to-date migration that reflects all database models.

¹<https://swagger.io/specification/>

```
docker-compose -f docker_compose_kafka.yml up
docker-compose -f docker_compose_metrics.yml up
docker-compose -f docker_compose_modules.yml up
```

4.4 Metrics

Prometheus was chosen as a tool for monitoring multiple services. Prometheus is collecting metrics by HTTP requests directed to the exporter metrics endpoint. Thus, the application has to implement an exposed metric endpoint in order to become an exporter. This can be done by using Prometheus client libraries, which have already implemented everything needed and provide a simple interface for users. Prometheus client library has four core metrics types, which are Counter, Gauge, Histogram, and Summary. The service monitoring uses the first three of the mentioned types. The defined metric can have various labels. These labels are used to differentiate the characteristics of measured metrics. The brief description of the first three metric types with use cases from core modules follows.

- **Counter:** The counter represents a single number, which can only be increased or reset to zero. This metric type is used in *address_pooler* to represent the total number of found new addresses.
- **Gauge:** This metric type represents a single number the same as the counter, but it can arbitrarily go down or up. The *node_watcher* uses it to represent the current count of active peers. Similarly, *address_publisher* is using it for the actual queue size.
- **Histogram:** The histogram type is counting observations and adding them to buckets. This metric type is used in the *address_publisher* to observe how long the RPC request took. Different durations of the requests are sorted into buckets, which allows us to make queries like how many requests took less than 1 second.

The Prometheus scrapes only exporters, which are listed with rules inside its configuration file. This configuration file is using YAML format, and this is how the set of rules for *nodes_watcher* exporter looks.

```
- job_name: 'node_watcher'
  scrape_interval: 1s
  static_configs:
    - targets: ['host.docker.internal:8002']
```

Prometheus scrapes each of the core modules exporters every second to gain precise insight over their work.

Chapter 5

Testing

The goal of this chapter is to test the implemented platform and validate the obtained results. Validating results is a crucial task in order to find out if the implemented platform will be any good. The database is filled with values from the platform, which did run over about one week.

The testing chapter divides into several parts. Firstly, the platform behavior during testing is analyzed, following with the evaluation of hardware requirements of Bitcoin Client, with increasing connections. Then, the section sums up the gained data about the node, addresses, and nodes activities. The chapter ends by comparing the obtained data with the other solution named Bitnodes.

5.1 Bitcoin Client Behaviour

The platform discovers new addresses by thousands in a few seconds after the start. Repeating the process of gathering new addresses every minute leads to the saturation of more and more addresses. Requested Bitcoin clients usually run out of new addresses within a few hours after the start. After the biggest amount of active connections is achieved, which is often after the first few hours because the significant amount of addresses is already collected.

Then turning point happens, and new addresses are more consumed than discovered. Connections are slowly decreasing since then. This can be observed in Figure 5.3. The saturation of addresses per requested client is in Figure 5.1. The total number of discovered addresses changing in time is shown in Figure 5.2. The second graph indicates that new addresses have logarithmic growth over a longer time rather than exponential as it was in the beginning.

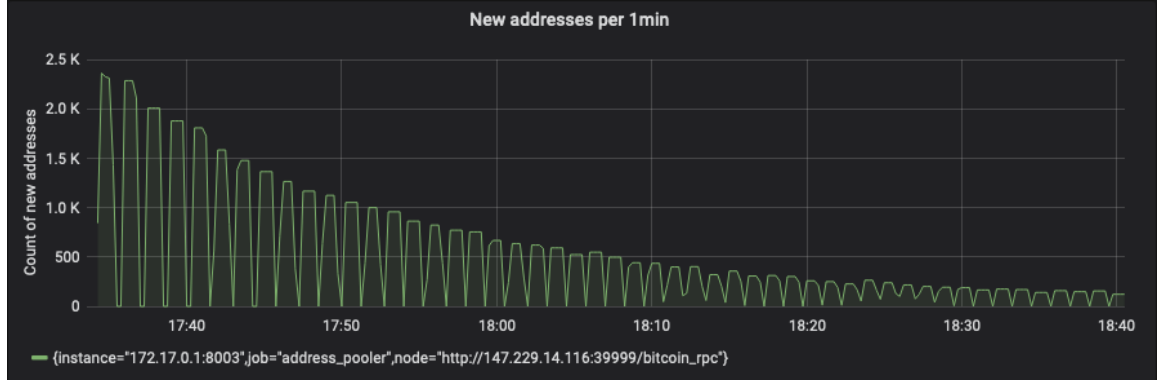


Figure 5.1: Gathered unique addresses per client

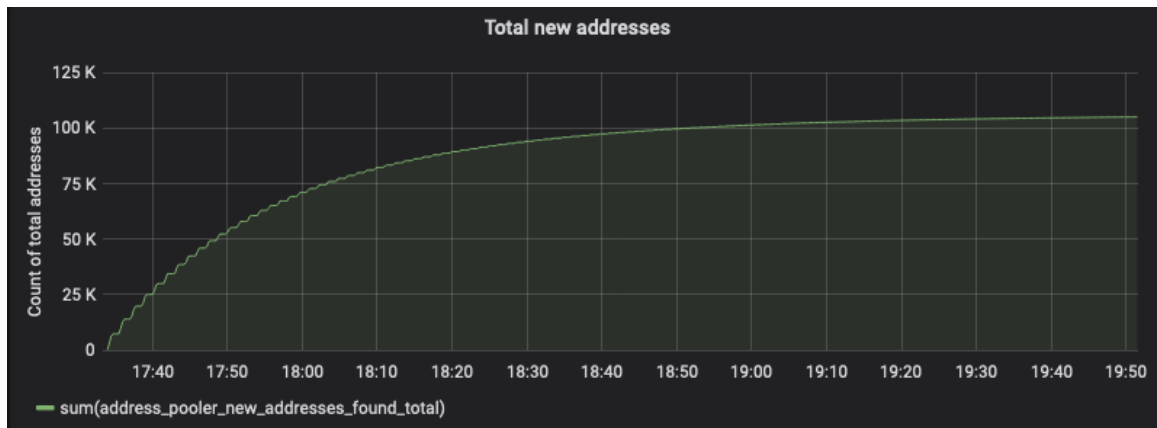


Figure 5.2: Gathered unique addresses per client

5.2 Bitcoin Client Requirements

I used the single Bitcoin Core client with version 0.19.1. The client run in prune mode with owning only the last 2000 blocks to lower the memory requirements of held blockchain. The number of client RPC threads was set to 30.

The implemented solution is using Bitcoin Core client in a way where limitations like insufficient RAM, CPU, or open sockets can occur. This is because the client is not primarily designed for several thousand active connections at one time but rather hundreds at maximum. However, the implementation of the client is capable of doing it.

The maximum amount of allocated resources shows in Figure A.1. The client is allocating more resources because of the increase in the new active connections. The highest peak in the graph is the time when the client reached the most active connections during monitoring by far. It was about 6700 active connections, as it shown in Figure 5.3. At that point, the client has allocated about 6.5 gigabytes of RAM and is having about 5500 open sockets.

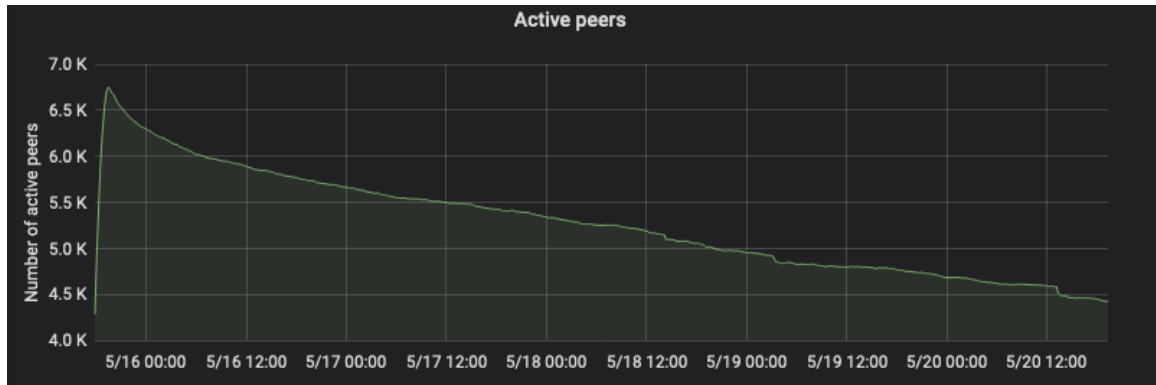


Figure 5.3: Active peers within days

In Figure A.2, it can be seen how are the resources changing since the peak within four days. Less active connections caused less open sockets, but the CPU and RAM values remained almost the same as at the highest peak. It means that the client is not giving up on hoarded resources, even the number of active connections decreased by hundreds over days. This can indicate many possible scenarios about what the client is doing. The client may need a longer time to give up resources, or it can be the first warning of poorly implemented resource management.

5.3 Summary of obtained data

150550 unique combination of IP address and port were collected as the result of monitoring, as can be seen in Listing 5.1.

```
select count(*) from ip_pool;

count
-----
150550
```

Listing 5.1: Count of all addresses

This is a vast number of addresses, but these are only addresses of potential nodes. I have to look at only those addresses which have assigned `node_id` to find out with how many unique nodes the client established connection. Firstly, I counted the unique nodes with addresses that occurred just once, which means these nodes have a static address. The client created 10124 active connections with nodes with a unique IP address, as can be seen as a result of SQL query in Listing 5.2.


```

select count(*) from
  (select ip
   from ip_pool
   where node_id notnull
   group by ip_pool.ip
   having count(ip) = 1
   ) unique_nodes_addresses;

count
-----
10124

```

Listing 5.2: Count of nodes with unique address

The count of nodes that have the IP address, which is assigned with at least one other node but has a different port is in Listing 5.3. The number of nodes that is obtained by the previously mentioned query is three times higher than that of nodes with unique IP addresses. Thus, the number of unique nodes with the same IP address has to be counted differently because of **possible duplicates**. The nodes with the same IP address and the different port numbers could be the same node just behind network address translation (NAT), which causes the changes of port numbers.

```

select sum(ip_count) from
  (select count(*) ip_count
   from ip_pool
   where ip_pool.node_id notnull
   group by ip having count(ip) > 1
   ) same_ips_count;

count
-----
31657

```

Listing 5.3: Count of nodes with same address

Saved **metadata** about given nodes are used in order to come up with this number.

The constructed query in Listing 5.4 is more complex and contains 3 subqueries. The query achieves wanted result by finding all possible node_ids grouped by IP address, and then counts of distinct user_agents are summed, these user_agents belongs to found node_ids grouped by IP addresses.

```

select sum(f.different_agents_by_ip_count) from (
  select count(distinct user_agent) different_agents_by_ip_count
  from node right join
    (select node_id, ip from ip_pool where node_id notnull and ip in
      (select ip from ip_pool group by ip having count(ip) > 1)
    )as a on id = a.node_id
  group by ip
) f;

sum
-----
4334

```

Listing 5.4: Count of unique behind NAT

The estimated number of unique nodes behind NAT is **4334**. These nodes were distinguished from duplicates only by user_agent metadata. The more precise result could be obtained by considering all metadata, but that could lead to a more complex query.

The total count of unique connected nodes is equal to the sum of unique nodes with static IP and estimated unique nodes behind NAT. Thus, the total count of discovered unique nodes estimates at **14 458**.

5.4 Discovery of my node

I run one other light node during the testing. The node that was creating connections did not know anything about the second node at his start. The purpose of this test will be to answer whether the platform can discover my other node. The second question is, if the platform is capable of discovering the second node run by me then how long it will take to create active connection.

My second node started at 2020-05-22 11:00:20. A few minutes later, I checked the database, the used query and response is shown in Listing 5.5.

```

select inserted, start_of_activity
from node_activity
join node n on node_activity.node_id = n.id
join ip_pool ip on n.id = ip.node_id
where ip = '37.205.14.85' and port = 8333

      inserted | start_of_activity
-----+-----
2020-05-22 11:01:53.119699+00 | 2020-05-22 11:01:55+00

```

Listing 5.5: Discovery of my other node

The testing node address appeared in the platform database **one minute and thirty seconds** later, after the node start. The active connection was established two seconds later after the address was spotted. This proves that the platform is able to discover and monitor new active nodes but under certain circumstances.

The discovering of new addresses depends mostly on communication between nodes in the network. Thus, it is hard to predict how long it will take for other nodes in the network to propagate the new addresses. In this testing case, it took less than 2 minutes. The other circumstance is that the newly discovered node has to be active. If the testing node went down earlier than it was discovered, the monitoring of this node would not be possible.

5.5 Validating results

I used one other source of Bitcoin nodes addresses to validate my achieved results. One of the open-source solutions for monitoring the Bitcoin network is called Bitnodes. The official site of Bitnodes is on this link¹.

Bitnodes uses a protocol level approach to find all nodes. This means there is a script with implemented Bitcoin client behavior to disguised itself as one of the nodes. After the script connects to nodes, it uses *getaddr* message to pull addresses directly from nodes. Thus, Bitnodes' solution is utterly dependent on the used version of the protocol. The new protocol releases can lead to the inability of communication from the script side with nodes that are using the newly released protocol. The main differences between using client and custom made script are depicted in the conclusion chapter.

Back to the testing, Bitnodes exposes results as 5 minutes snapshots through HTTP API. The different snapshots are available behind the `/api/v1/snapshots/{timestamp}` endpoint. I have created simple Python script shown in Listing 5.6 in order to compare six days of monitoring between my platform and Bitnodes. The test is based only on comparing nodes addresses without any additional metadata about nodes. One Bitnodes snapshot is representing the whole day in this testing.

Firstly, I parsed and created the set of nodes addresses from various Bitnodes snapshots. Then, the script iterates the obtained set of addresses and requests my API with each address. I used `/nodes/{ip_with_port}` endpoint from my created API. This endpoint returns information about the node with an address that was provided as a parameter or returns Not Found HTTP status code, as mentioned in Subsection 4.3.4. So if my API returned HTTP OK status code, which is represented by the number 200, then node would be found by my platform as well.

My platform has discovered **14 458** nodes in contrast to **12 221** from Bitnodes. The difference is caused by running exposed Bitcoin Client, which means other disguised scripts similar to Bitnodes solution with unique metadata can connect to the client and trick him into thinking that it is communicating with other bitcoin nodes. Thus, the total number of unique nodes is increased.

The **7687** nodes were discovered by Bitnodes as well as by my platform. The missing **4553** active connections is a fairly huge difference at first glance. It requires deeper investigation to tackle the root of the problem, so let us dive in. Firstly, I repeated same script shown in Listing 5.6 as above but hitting `/address` endpoint in my API instead of `node` endpoint. The difference is that the results from the `address` endpoint would tell us if the missing addresses were seen by my platform regardless of the assigned nodes to that address. The result of the script was **9053** seen addresses of the total number **12 221**. There were still 3168 addresses missing. I thought that already mentioned problems could cause it as insufficient address propagation by other participants in the network. As

¹<https://bitnodes.io/>

I scrolled down through logs of my API, which are in Figure 5.4, I saw the thing that many of the missing addresses had in common.

```
timestamps = [1589640727,1589713317,1589795379,1589866651,158993908, ...]
addresses = {
    requests.get(f"https://bitnodes.io/api/v1/snapshots/{i}/")
    .json()["nodes"]
    .keys()
    for i in timestamps
}
compare_snaps = [
    True
    if requests.get(f"http://localhost:8000/node/{i}").status_code == 200
    else False
    for i in addresses
]
print("Bitnodes addresses count: ",len(addresses))
print("Addresses in common: ", sum(compare_snaps))

Bitnodes addresses count: 12221
Addresses in common: 7687
```

Listing 5.6: Script that compares nodes addresses with Bitnodes

```
INFO: 127.0.0.1:63772 - "GET /address/zcgavvceh53zkaru.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63774 - "GET /address/4rgc72iegnq5nwdu.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63776 - "GET /address/e7rjoluxbw3oce2.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63778 - "GET /address/bevvp6uouq266tkd.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63780 - "GET /address/148.251.20.79:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63782 - "GET /address/qrtaxviujezc6ro.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63784 - "GET /address/fayauxoqg6ux7azj.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63786 - "GET /address/216.186.250.53:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63788 - "GET /address/[2a01:4f8:190:8361::2]:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63790 - "GET /address/173.255.209.85:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63792 - "GET /address/2nyrqh7ake2wvvyu.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63794 - "GET /address/18.220.91.12:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63796 - "GET /address/159.100.245.237:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63798 - "GET /address/209.188.21.65:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63800 - "GET /address/djhudr2ovodzeyv7.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63802 - "GET /address/23.238.48.248:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63804 - "GET /address/[2001:470:7:b74::2]:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63806 - "GET /address/148.251.178.72:8433 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63808 - "GET /address/172.90.117.120:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63810 - "GET /address/82.13.20.247:8333 HTTP/1.1" 200 OK
INFO: 127.0.0.1:63812 - "GET /address/4dgd5ml6mxbifs3.onion:8333 HTTP/1.1" 404 Not Found
INFO: 127.0.0.1:63814 - "GET /address/jspxanv3v44pvg6w.onion:8333 HTTP/1.1" 404 Not Found
```

Figure 5.4: Log from the API during the address comparison process.

The missing addresses had **.onion** suffix, which means that these addresses belong to the special onion service domain. The onion service is only accessible via the Tor network, and thus, these addresses are not accessible via the public Internet. I looked up all the

provided addresses from Bitnodes with **.onion** suffix and check their results in lastly run script. As a result, only two onion addresses were seen by the platform, and the rest **2618** addresses were never seen. All of this investigation leads to the fact that there were only missing **550** addresses reachable via the public Internet, and all other addresses were in the platform database.

It is far better result than I expected when comparing the use of the one Bitcoin client with restricted scope to the RPC methods against the solution as Bitnodes, which pulls addresses directly from the nodes through the solution tailored to the given bitcoin protocol. With deducted onion addresses and ones that were not seen, the platform did not establish a 1382 potential connection that Bitnodes did. One of the reasons for this may be that the nodes were already down when clients tried to establish a connection or the nodes went down, and a few hours later up again. Other factors could play a role here, so I assume this is a space for the platform future improvements. One of the possible solutions for mentioned cases would be rescheduling already tried addresses, which were unavailable at a specific time but might later be able.

Chapter 6

Conclusion

This bachelor thesis aimed to create the platform for monitoring nodes of the Bitcoin network and gathering metadata about found nodes. It was necessary to study the principles of cryptocurrencies and their communication protocols. Gained knowledge allowed me to understand better the concepts of cryptocurrencies and communication between nodes in the network. All the theoretical prerequisites are in Chapter 2.

Then, I analysed the interesting metadata (user_agent, protocol version, services) about nodes. These metadata can help to uniquely identify nodes behind NAT as described in Section 3.1 and proved in Listing 5.4. Furthermore, I have designed the platform for discovering and monitoring the nodes in the network. The platform is based solely on using API of the official Bitcoin client to retrieve all information about the nodes. The client is used without any further modification. The proposed design of the platform is in Chapter 3.

The implementation insights are described in Chapter 4. The implemented platform is able to estimate the size of the Bitcoin network and to monitor nodes at the given time. The nodes monitoring involves gathering metadata about found nodes and creating activity records about how long they are participating in the network, as described in the testing of the platform in Chapter 5.

Testing of the platform includes benchmarks of the Bitcoin client in Section 5.2, evaluation of obtained data from one week-long monitoring session in Section 5.3. The platform discovered and connected to the second node run by me within 2 minutes, as described in Section 5.4. The validation was based on comparing results with the Bitnodes solution in Section 5.5. The Bitnodes solution uses script tailored to the Bitcoin protocol, which can result in future incompatibilities with new protocol versions. Results from the created platform were successfully validated and proved that the platform based on only using the Bitcoin client is comparable good as custom protocol solutions even the client API calls have limited scope of use.

The created platform could be extended into the support of more cryptocurrency networks in the future.

Bibliography

- [1] BINANCE.LTD.. *What are nodes?* [online]. Binance, 2018 [cit. 2019-12-16]. Available at: <https://www.binance.vision/blockchain/what-are-nodes>.
- [2] BITCOIN.ORG. [cit. 2020-01-11]. Available at: <https://bitcoin.org/en/p2p-network-guide#initial-block-download>.
- [3] DEVAWRITES. *Lightweight node*. January 2018 [cit. 2019-01-11]. Available at: https://en.bitcoin.it/wiki/Lightweight_node.
- [4] FRANKENFIELD, J. *Consensus Mechanism (Cryptocurrency)* [online]. Investopedia, 2019 [cit. 2019-12-30]. Available at: <https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp>.
- [5] GERVAIS, A., KARAME, G. O., WÜST, K., GLYKANTZIS, V., RITZDORF, H. et al. *On the Security and Performance of Proof of Work Blockchains*. New York, NY, USA: Association for Computing Machinery, 2016. Available at: <https://doi.org/10.1145/2976749.2978341>.
- [6] GROUP, T. E. *Who is Satoshi Nakamoto?* The Economist Newspaper Limited, November 2015 [cit. 2019-12-30]. Available at: <https://www.economist.com/blogs/economist-explains/2015/11/economist-explains-1>.
- [7] HEARN, C. M. *How XRP Stacks Up Against Other Digital Assets* [online]. October 2012 [cit. 2019-1-19]. Available at: <https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki>.
- [8] HILEMAN, G. and RAUCHS, M. Global cryptocurrency benchmarking study. *Cambridge Centre for Alternative Finance*. Cambridge Centre for Alternative Finance. [online]. 2017, vol. 33, [cit. 2019-12-30]. Available at: <https://cdn.crowdfundinsider.com/wp-content/uploads/2017/04/Global-Cryptocurrency-Benchmarking-Study.pdf>.
- [9] IANSITI, M. and LAKHANI, K. R. *The Truth About Blockchain* [online]. Harvard University, January 2017 [cit. 2019-12-30]. Available at: https://enterpriseproject.com/sites/default/files/the_truth_about_blockchain.pdf.
- [10] JOHNUAQ. *Remote procedure call*. November 2019 [cit. 2019-12-30]. Available at: https://en.wikipedia.org/wiki/Remote_procedure_call.
- [11] KARAME, G. and ANDROULAKI, E. *Bitcoin and Blockchain Security*. Artech House Publishers, 2016. 51 p. Artech House information security and privacy series. Available at: <https://books.google.sk/books?id=YYSuDGAAQBAJ>. ISBN 9781630814335.

- [12] MAULL, R., GODSIFF, P., MULLIGAN, C., BROWN, A. and KEWELL, B. *Distributed ledger technology: Applications and implications* [online]. FINRA, 2017 [cit. 2019-12-30]. Available at: <http://epubs.surrey.ac.uk/814158/1/Distributed%20Ledger%20Technology%20Applications%20and%20Implications.docx>.
- [13] NAKAMOTO, S. *Bitcoin: A peer-to-peer electronic cash system*. 2009 [cit. 2019-1-20]. Available at: <http://www.bitcoin.org/bitcoin.pdf>.
- [14] PAPPALARDO, G., DI MATTEO, T. and CALDARELLI, G. *Blockchain inefficiency in the Bitcoin peers network* [online]. Springer Berlin Heidelberg, September 2018 [cit. 2019-12-30]. Available at: <https://doi.org/10.1140/epjds/s13688-018-0159-3>.
- [15] RIPPLE.ORG. *How XRP Stacks Up Against Other Digital Assets* [online]. December 2017 [cit. 2019-1-19]. Available at: <https://ripple.com/xrp/xrp-stacks-digital-assets/>.

Appendix A

Figures



Figure A.1: Allocated resources by Bitcoin Client at its peak with active connections

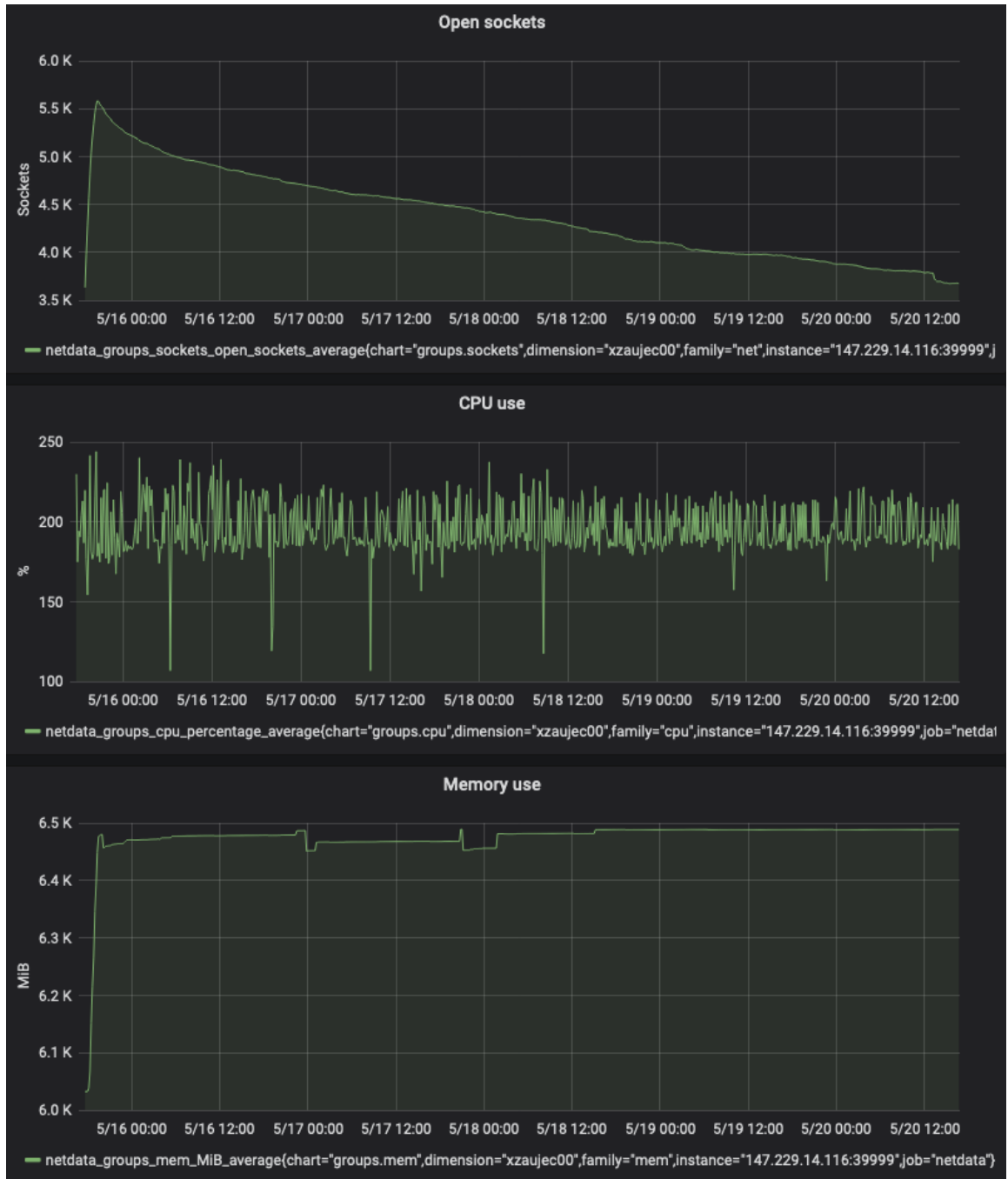


Figure A.2: Allocated resources by Bitcoin Client within several days

[illegible]

Figure A.3: Begin of the nodes communication

Appendix B

CD contents

The CD has same content as my git repository¹ on branch **origin/bachelor__thesis**. The first level of the CD file structure is following:

- **xzauje00.pdf** – This thesis
- **app** – All source code of the platform
- **alembic** – Database migrations
- **alembic.ini** – Database migrations configuration
- **.env.example** – Example of proper .env file
- **Dockerfile** – Docker container for one module
- **docker-compose-kafka.yml** – Docker deployment for Kafka
- **docker-compose-modules.yml** – Docker deployment for modules
- **docker-compose-metrics.yml** – Docker deployment for metrics
- **prometheus.yml** – Prometheus scraping configuration
- **requirements.txt** – Python requirements for modules
- **crypto_watch_db_backup_with_inserts.tar** – Database snapshot done with pg_dump 12.1
- **README.md** – Short description of using with examples
- **LICENSE.md** – MIT license

¹https://github.com/fruit098/crypto_monitor