TÉCNICO LISBOA

Behavior characterization in cryptocurrency networks

David Vavřička

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisors: Prof. Miguel Ângelo Marques de Matos
Prof. João Pedro Faria Mendonça Barreto

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Nuno Manuel Ribeiro Preguiça

May 2019

Acknowledgments

I would like to thank my supervisors, professor Miguel Matos, professor João Barreto and professor Paulo Silva, for their support, patience, advice and constant guidance during the development of this thesis.

This work was partially supported by Fundo Europeu de Desenvolvimento Regional (FEDER) through Programa Operacional Regional de Lisboa and by Fundação para a Ciência e Tecnologia (FCT) through project with reference LISBOA-01-0145-FEDER-031456.

Resumo

Ao longo dos últimos anos, os sistemas de criptomoedas descentralizadas têm vindo a ganhar um rápido crescimento e interesse não só por parte dos entusiastas tecnológicos, mas também do público geral. Tal interesse deve-se ao potencial para revolucionar os métodos de pagamento atuais, uma vez que as criptomoedas pretendem eliminar os bancos e outras entidades centrais – que atuam como mediadores de confiança de operações financeiras – fornecendo uma tecnologia que inerentemente garante uma execução confiável e honesta de transações.

São baseados nas redes ponto-a-ponto que permitem a disseminação de novas transações, ou a distribuição do estado mais recente da moeda entre toda a plataforma. No entanto, como foi observado, há a possibilidade de ocorrerem atrasos na propagação de dados, o que pode causar um estado inconsistente temporário dos nodos em zonas da rede da moeda que são geograficamente distantes; causando uma maior vulnerabilidade a ataques para além de outros efeitos indesejáveis. De maneira a reduzir essas inconveniências, deve ser implementado um protocolo de disseminação robusto e desenhado para uma boa interconexão entre os pontos, assim como baixos tempos de propagação mesmo perante colaboradores distantes.

Este trabalho explora o protocolo de rede do Ethereum, que é uma das mais proeminentes e inovadoras plataformas baseadas em blockchain, através da medição de várias métricas e características da rede. O objetivo das nossas medições é identificar o comportamento da rede, descobrir os pontos fracos, e com isso permitir melhorar os sistemas existentes. Fazemos deploy de um conjunto de nodos geograficamente distribuídos baseado numa implementação open-source do protocolo Ethereum. Modificamos os nodos de modo a ficarem à escuta do tráfego na rede principal do protocolo e fazemos medições no espaço de um mês.

Tanto quanto sabemos, somos os primeiros a medir a propagação dos atrasos de blocos e transações entre pontos Ethereum geograficamente dispersos. Neste aspeto, observamos um desempenho significativamente superior em comparação com a criptomoeda competidora Bitcoin. Para além disso, replicamos várias medidas de 2017 e mostramos como a rede Ethereum mudou e evoluiu desde então. No fim, avaliamos e comparamos os resultados com o trabalho de outros, e damos possíveis sugestões de como melhorar o protocolo.

Palavras-chave: Criptomoedas, Ethereum, Redes P2P, Sistemas Descentralizados

Abstract

Over recent years, decentralized cryptocurrencies enjoyed a rapid growth and gathered remarkable interest no longer just from technological enthusiasts, but from the general public as well. This is due to their potential to revolutionize the current payment methods with the ambition to eliminate the need for banks or other centralized entities – acting as trusted mediators of financial operations – by guaranteeing honest and proper execution of transactions inherently by the technology itself.

They are based on peer-to-peer networks allowing for dissemination of newly created transactions, or distribution of the most recent currency's state among the whole platform. Nonetheless, as it has been observed, possible delays of data propagation may lead to a temporarily inconsistent state of nodes in geographically distant parts of the currency's network; causing the increased vulnerability to attacks besides other undesirable effects. To significantly lower the chances of such inconveniences, a well designed dissemination protocol should aim for a good interconnection of peers and low propagation times even among distant collaborators.

This work explores the network protocol of Ethereum, which is one of the most prominent and innovative blockchain-based platforms, by measuring various network metrics and characteristics. The goal of our measurements is to identify the network's behavior, find out the weak points, and enable so several improvements to the existing system.

We deploy a set of geographically distributed nodes based on an open-source implementation of the Ethereum protocol. We modify the nodes to listen to the network traffic on the main network of the protocol and perform one-month long measurement.

To the best of our knowledge, we are the first to measure the propagation delay of blocks and transactions between geographically dispersed Ethereum peers. In this aspect, we observe the significantly better performance in comparison with the competing cryptocurrency Bitcoin. On top of that, we replicate several measurements from 2017 and show, how the Ethereum network changed and evolved since then. In the end, we evaluate and compare the results with works of others and provide with possible suggestions about how to improve the protocol.

Keywords: Cryptocurrencies, Ethereum, P2P Networks, Decentralized Systems

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

In late 2008, the author or authors hiding under the pseudonym "Satoshi Nakamoto" posted a paper [Nak08] on a cryptography mailing list, which started a revolution in the payment industry. This paper introduced the innovative distributed ledger technology that would later become known as blockchain. This technology together with usage of cryptographic primitives and a proof-of-work consensus algorithm, which was also introduced by Nakamoto, allowed for the creation of Bitcoin; the first global and truly decentralized cryptocurrency.

Bitcoin inspired over time not only the creation of other cryptocurrencies but also the creation of general purpose blockchain based decentralized systems. These systems have the goal of potentially decentralizing the current state of the Internet, which is predominantly based on centralized web services.

Ethereum, which is a platform for developing and implementing decentralized applications, represents one example of such systems. Its major novelty lies in the introduction of a Turing-complete execution environment on client nodes allowing for applications more complex than simple money transfers. These applications are known as smart contracts and include, for example digital wallets, various games, or multiparty value exchanging contracts. Moreover, Ethereum includes its own currency called Ether which is the second most valued cryptocurrency right after Bitcoin.

Despite all the positive remarks about Ethereum mentioned above, the Ethereum's underlying protocol, as any new technology, is far from being perfect and needs some time to mature. The most critical issue is rooted in the current approach of building blockchain systems, where every peer of the network processes all information of the whole platform, which causes the impossibility of the system to scale [CDE+16]. This inherently makes such systems not suitable for mass usage.

Another complex part of the Ethereum platform is the underlying peer-to-peer network protocol, which must support the interconnection of a large number of peers spread all over the

world, prevent the creation of network partitions, and allow for a smooth transfer of all the data needed in the system. Ethereum went live on 30 July 2015 and so far the network performed well and showed to be able to cope with the relatively large number of peers. Recent estimations say that there are around 15 000 peers [KMM+18]. Nonetheless, the overall network is poorly studied. We are missing many metrics describing the network behavior, most importantly about the information propagation between individual peers. Such a study could enable find possible, and until now hidden, pitfalls and shortcomings of the protocol, and to observe possible aspects to be optimized.

For Bitcoin, which has been out for longer, there exist many more analyses and studies. In the network behavior context, there is one highly cited paper [DW13] measuring, among other metrics, the propagation delay of blocks between geographically dispersed nodes in the Bitcoin network. The study concludes that the propagation delay of blocks is the primary cause of blockchain forks. As far as we are aware, no one had measured this metric in the Ethereum network before. We decided to measure this metric – among many others – in our work in order to see how things differ in Ethereum, which follows a proof-of-work based consensus algorithm similar to Bitcoin's.

In Section 1.1, we present the remaining goals of this thesis.

1.1 Goals

Previous work [WGP+] from 2017 studied various properties of the Ethereum platform, such as what impact the user-defined variables "gas price" and "gas limit" have on transaction inclusion, the commit time of transactions or the ratio of blockchain forks. We replicate and verify their measurements in order to observe how the properties changed over the last two years. Moreover, we take attention at details, that the authors of [WGP+] omitted, such as the relation between fork lengths and the probability of being referenced in main blocks. On top of that, we measure various metrics regarding the forked blocks, e.g. the degree of transaction reordering, and add several other metrics, such as the degree of redundant receptions of new block announcements, with the aim to observe the network traffic overhead.

Since the measurements performed in [WGP+] were captured all by one statical monitoring node, our novelty lies in deploying a set of geographically dispersed monitoring nodes to examine whether there is a correlation between the metrics captured and the geographical position of particular nodes.

In order to perform the measurements, we develop a tool based on an already existing open-source implementation of Ethereum client which is able to connect to the main network and

collect the desired data. We deploy four these measuring instances that we place in North America, Western Asia, Western Europe and Central Europe.

1.2 Contributions

The contributions of this work can be enumerated as follows:

1. To the best of our knowledge, this work is the first to measure the propagation delays of blocks and transactions in the Ethereum platform.
2. We analyze how the Ethereum network evolved since 2017 by replicating measurements performed in [WGP+] from that year.
3. We find several interesting phenomena. We observe peers from the competing Ethereum classic platform propagating useless messages and flooding so the otherwise separate Non-classic Ethereum main network. We capture significant amount of empty blocks that are harmful for the platform. To mention one more out of many observations, we investigate how the transactions are reordered in forked blocks and argue, how this could be potentially leveraged by the protocol.
4. Based on the measured results, we discuss what is the current condition of the Ethereum platform from the network perspective.
5. We propose an idea about how to modify the Ethereum clients in order to perform more efficiently. Furthermore, we discuss potential high-level changes to the Ethereum platform.

1.3 Document Structure

The rest of this document consists of the following Chapters:

- Chapter 2 provides the technological background needed later in this document.
- Chapter 3 examines papers related to Bitcoin, Ethereum and blockchain systems.
- Chapter 4 describes the measurements performed in this work.
- Chapter 5 presents the measuring infrastructure and implementation details.
- Chapter 6 evaluates the results.
- Chapter 7 discusses the final achievements and finishes this document.

Chapter 2

Background

In this chapter, we provide a high-level overview of the blockchain systems and their properties such as the mining process or the proof-of-work consensus algorithm. Since this work deals with the network behavior in the Ethereum platform, we present the underlying network protocols, specifications and the particular elements which the network consists of. In the end, we give a brief introduction into Ethereum client implementations and the Geth client in particular.

2.1 Blockchain

Blockchain represents the backbone technology which allowed for the birth of the first widely used cryptocurrency – Bitcoin. Before that, all the attempts at building a decentralized cryptocurrency failed due to the lack of a non-centralized mechanism which would guarantee to prevent double-spending of funds. Classical cash, thanks to its physical form, is inherently not vulnerable to double spending; once paid, the cash changes hand and the original payer cannot pay with the same cash anymore. In the case of cryptocurrencies, where the coins are represented by digital strings, if implemented naively, one could simply copy such a string to enhance their riches. Bitcoin solves that by assigning a definite, unique identity to each coin and storing a history of all spendings of all coins in the circulation on a blockchain public ledger.

Later, as blockchain became more popular, it found its usage in various kinds of platforms; not just those related to cryptocurrencies. In general, blockchain technology allows building systems that permit untrusted parties to reach an agreement on a common digital history without the need for a trusted intermediary.

Under the hood, blockchain is a tree consisting of blocks (data structure containing transactions), where every block stores a cryptographic hash of the preceding block. A blockchain system can be seen as a replicated state machine, meaning that every peer of the platform pos-

sesses a local copy of the entire blockchain, which stores information about the platform's state. The participants of the network must obviously agree on a common state of the system to assure execution of transactions in the same order by all peers. There exist various consensus algorithms for this purpose. The currently predominant one, used both in Bitcoin and Ethereum, is called proof-of-work and is discussed in more detail in the following Section 2.2.

2.2 Mining

Mining represents a process in which the subset of platform participants, called miners, validate transactions and orderly collect them in newly created blocks. Individual miners compete with each other, trying to mine blocks as fast as possible as only the fastest one is rewarded. Mining rewards a successful miner with a fixed amount of Ether coins, along with transaction fees for every transaction included in the newly mined block.

Miners, in addition to validating transactions, must perform the technique called proof-of-work (PoW) which consists of resolving computationally expensive cryptopuzzles. Under the hood, PoW consists of repetitive hashing of a string consisting of the block's data and a nonce. A miner tries to perform the computations for a tremendous number of nonces, until the final produced hash is less than a certain threshold – determined by the difficulty of the network. Only then, the block is valid and can be propagated to the network.

The fact that PoW is very hard to perform guarantees the security of the platform, since it makes it resistant to being flooded by false blocks and assures the near impossibility of eventual future modifications of previously confirmed parts of the blockchain. Furthermore, it makes miners spend some time on performing the computations, allowing the rest of the network to have enough time to synchronize.

Due to the computational complexity of mining in the Ethereum network, it is practically impossible for an individual running a personal computer to find new blocks. For this reason, mining is performed either on powerful clusters or, even more often, on mining pools. The latter are discussed in the following section.

2.2.1 Mining Pools

A typical mining pool consists of a private network running a command and control server distributing a workload among individual miners. The workload consists of a Merkle root of a set of transactions and a range of nonces to search through.

The individual participants of mining pools skip the validation phase and solve just a "sub-puzzle", of the full cryptographic puzzle, with a lower difficulty. When a participating miner

finds a solution for the puzzle, he echoes the server, which uses the information for generating the final block. After that he tries to propagate it as soon as possible through its gateways to the Ethereum network. The miners are later justly rewarded according to the computational resources they spent. The pool typically charges a fee of around one percent[1].

2.2.2 Pool Distribution in Ethereum

Figure 2.1 demonstrates the rate of mining decentralization on the Ethereum platform. The results show that around 90% of the mining power is controlled by only 12 mining pools. On average, more than 48% of the weekly power is shared by only two Ethereum mining pools. This represents a big threat to the whole platform as the two dominant mining pools could gain control over the whole network by exploiting the famous 51% attack, which happens when a miner or group of cooperating miners access more computational power than all the honest miners.

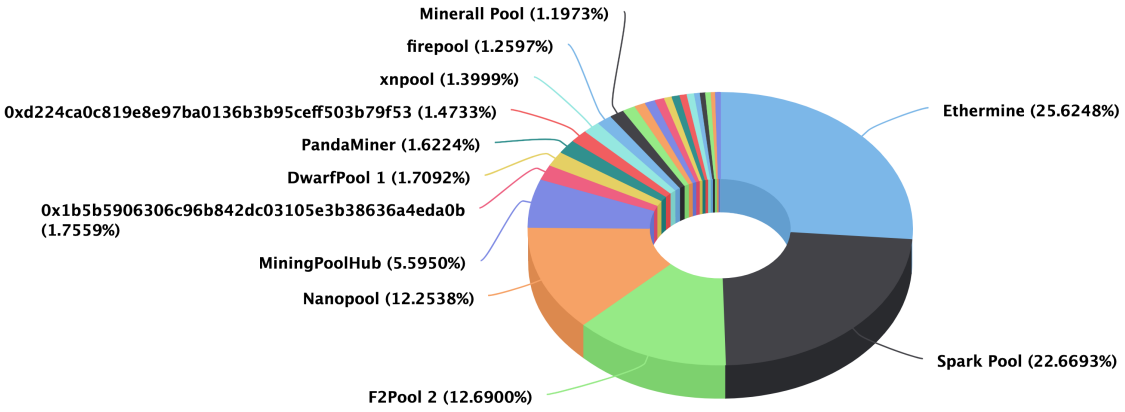


Figure 2.1: The Ethereum mining pools' distribution in the 7 preceding days as of April 25 2019. (etherscan.io)

2.3 Ethereum Basics

Whereas Bitcoin's usage was limited to exchange and store Bitcoin cryptocurrency, Ethereum not only supports the exchange of its currency (Ether) but also allows to create smart contracts and decentralized applications for general purpose. That is achieved due to the support of a Turing-complete programming language.

Before presenting the technical details of Ethereum, we feel obligated to give an introduction to smart contracts and decentralized applications, which represent the core building blocks of the Ethereum platform.

[1] https://www.cryptocompare.com/mining/#/pools

2.3.1 Smart Contracts

A smart contract is a distributed program capable of execution when certain predetermined conditions are met. This allows peers who do not trust each other to make contracts and agreements without requiring any central authority nor other intermediaries. Cautious users may analyze the expected contract behavior, as the source codes are publicly available. Trust in correct execution is assured by the protocol itself – e.g. Ethereum.

Another possible view at smart contracts presents Vukolić in [Vuk15]: "Indeed, a smart contract can be modeled as a state machine, and its consistent execution across multiple nodes in a distributed environment can be achieved using state machine replication".

For Ethereum there exist various different high-level languages for programming smart contracts, to mention a few – Vyper[2] (syntactically similar to python), LLL[3] (Lisp-like language) or Solidity[4] (JavaScript-like), out of which all eventually compile to the same bytecode.

In order to give readers a better idea how a smart contract looks like, below we show a sample contract written in Solidity, which is the most widespread language of the Ethereum platform at the time of writing this document.

Smart Contract Example

```
1  contract Faucet {
2      // Give out ether to anyone who asks
3      function withdraw(uint withdraw_amount) public {
4          // Limit withdrawal amount
5          require(withdraw_amount <= 100000000000000000);
6          // Send the amount to the address that requested it
7          msg.sender.transfer(withdraw_amount);
8      }
9      // Accept any incoming Ether
10     function () public payable {}
11 }
```

Listing 2.1: Solidity Smart Contract

In Listing 2.1 we present the code of a simple wallet that allows for withdrawals as well as for refills of Ether. The code is supposed for academic purposes only and should not be used in practice. This is because in a real world scenario we would not like to allow everyone to withdraw from our wallet. This example aims to be as simple as possible in order to show the very basic smart contract functions.

[2]https://github.com/ethereum/vyper
[3]https://github.com/ethereum/solidity/tree/develop/liblll
[4]https://github.com/ethereum/solidity/

The first line declares a contract object that we named Faucet. On the third line, we define a function called withdraw where the *public* keyword allows all contracts to call this function. On the fifth line we use the built-in Solidity *require* function to test that the function's input parameter does not exceed a certain limit of Ether. If the condition is not met, the *require* function terminates the contract execution and raises an exception.

The *msg* object on the seventh line is one of the global variables which are mainly used to provide information about the blockchain or as general-use utility functions. In our case, we access through it the sender that triggered this contract execution. The *transfer* function, as the name suggests, sends him an amount of Ether (in wei) specified in the argument.

Finally, on the 10th line we define a second function, in this case a public no name function. In Solidity, there can only be one no name function per contract, and they are meant to capture all transactions sent to to the contract that have no data – i.e. Ether transfers – or those transactions that do not call any valid function of the contract. The *payable* annotation makes all the currency sent to it to be owned by the contract.

Deploying a Contract

A contract must be first compiled into a bytecode before deploying on the blockchain. For each Ethereum language there exist numerous compilers either as standalone executables or as a part of Integrated Development Environments (IDEs).

Then it remains to send a contract creation transaction with the compiled code in the init field of transaction to target address 0x0000000000000000000000000000000000000000 also known as the "zero address".

Layman users do not need to know these details as every solid IDE performs that automatically. The IDE or the console program used for sending such messages will then return the address of the newly deployed contract for further interaction.

2.3.2 Decentralized Applications

Decentralized Applications (Dapps) may be seen as classical programs with the difference that they are based on smart contracts and run on decentralized network platforms, rather than on a central source. Their biggest advantage is that they are highly fault-tolerant since they do not rely on a single point of failure.

Dapps in the Ethereum network are booming. At the moment, there are more than one thousand of them[5] and new ones are appearing by the day. Today, the most popular one is the

[5]https://www.stateofthedapps.com

Cryptokitties[6] game, in which users can collect and breed digital kittens. The game had become so popular, that it accounted for over 15% of total Ethereum network traffic in December 2017 causing a prolonged transaction commit time.

2.3.3 Basic Blocks

In this section, we introduce and define some of the basic elements of Ethereum.

Accounts

Accounts are stateful objects whose states – e.g. account balance – can be changed by sending transactions.

There are two types of Ethereum accounts: externally owned ones which are controlled by private keys – similar to those in Bitcoin – and contract accounts which are associated to smart contract code whose functions can be executed upon a transaction reception.

Transactions

In the Ethereum terminology, a transaction refers to the signed data package, assembled by externally owned accounts, which is later propagated over the P2P network to other nodes.

A transaction contains the following fields [Woo14]:

nonce: Total number of transactions sent from the sender's account.

gasPrice: The price per computational step.

gasLimit: The maximum gas limit to be spent during the execution of the transaction.

to: The recipient's address (empty if the transaction is used to create a contract).

value: The amount of Ether to be sent to the recipient.

v, r, s: The signature's parameters used to determine the sender of the transaction.

data: Input data of a contract which is used only if the transaction represents a message call.

init: Initial code for a contract which is used only if the transaction creates a new contract account.

[6] https://www.cryptokitties.co

Messages

Contracts communicate by using "messages" which are virtual objects that exist only in the Ethereum execution environment. A message is created as a result of a transaction calling the contract. They are never relayed over the network since they exist as a part of the Ethereum execution environment.

Blocks

The block data structure is formed by three main parts: the header, the list of uncle block headers, and the list of transactions included in the block.

The block header consists of the following fields [Woo14]:

parentHash: The hash of the parent block's header.

ommersHash: The hash of the list of uncle blocks' headers.

gasLimit: The maximum gas limit to be spent during execution of all transactions included in the block.

beneficiary: The account address that receives the mining reward and fees.

stateRoot: The hash of the state trie's root. The state trie is a merkle tree based data structure containing the global state about all accounts in the blockchain.

transactionsRoot: The hash of the transaction trie's root.

receiptsRoot: The hash of the root of the trie with receipts of transaction.

logsBloom: The bloom filter composed from indexable information contained in each log entry of each transaction in the transactions list.

difficulty: The mining difficulty of the block.

number: The block height.

gasLimit: The maximum gas limit to be spent during the execution of transactions included in the block.

gasUsed: The amount of gas used by executing the transactions included in the block.

timestamp: The creation timestamp of the block set by the miner.

extraData: An arbitrary byte array of 32 bytes that the miner wishes to append to the block.

mixHash: A hash computed by the PoW algorithm, which combined with the nonce, proves that a sufficient amount of computation has been carried out on this block.

nonce: A 64-bit number found by performing the PoW algorithm, which together with the mix-hash, proves that a sufficient amount of computation has been carried out on this block.

2.4 Ethereum Network Plane

The Ethereum network communication follows a peer-to-peer (P2P) approach and takes place over TCP with the only exception of UDP-based node discovery mechanism.

2.4.1 Network Protocols

The P2P communication between Ethereum clients is governed by the underlying DEVp2p Wire Protocol[7] and RLPx[8] cryptographic protocol suite providing a general-purpose transport interface. The RLPx allows for node discovery, transport of messages encrypted with AES256 or packets framing to support multiplexing multiple protocols over a single connection.

This protocol provides with a standardized API but the general behavior of peers on the network and their synchronizing strategies are implementation dependent.

2.4.2 Ethereum Protocol Properties

The Ethereum protocol aims for a block interval – which is the mean time it takes to mine a new block – between ten to twenty seconds. This is ensured by the mining difficulty property which is automatically and continually adjusted. The block size is limited by a property called block gas limit. Gas represents the internal pricing for running transactions and its price – in Ethers – is determined by the market. The amount of gas needed to run a transaction depends on the computational complexity, memory requirements and on the fee the sender is willing to pay for it.

The low block interval – ∼40 times shorter than in Bitcoin – allows for better response time, which is a desired property in cryptocurrencies, and for better throughput. On the other hand, short block intervals lead to an increased amount of stale blocks. Those are the blocks that would never become a part of the main blockchain. Huge stale block rates in classical Nakamoto's consensus impede individual miners, thus leading to mining centralization. They

[7] https://github.com/ethereum/wiki/wiki/ÐΞVp2p-Wire-Protocol
[8] https://github.com/ethereum/wiki/wiki/RLP

also weaken the absolute strength of the network against attacks, as the inventor and co-creator of Ethereum Vitalik Buterin explains in his blog post[9].

The GHOST Protocol

Ethereum, as a countermeasure against the unpleasant consequences of the short block interval, implements a modified version of the Greedy Heaviest Observed Subtree (GHOST) protocol [SZ15]. The original Nakamoto's consensus protocol [Nak08] considers the longest chain as the one which represents the current state of a blockchain network, whilst the GHOST protocol considers the heaviest one. In principle, the weight is calculated from the cumulative difficulty of the main chain and the uncle blocks up to x-th generation. The uncle blocks – also called ommer blocks – are stale blocks that do not represent a part of the main chain.

The Ethereum GHOST modification deviates from the original one in the fact that it rewards not just the blocks on the main chain, but the stale ones as well. This fact mitigates the centralization issue since even less powerful miners are justly rewarded for their efforts.

2.4.3 Ethereum Clients

In general, we distinguish between two basic types of peers: full nodes and light ones. Full nodes possess the entire blockchain, verify the validity of all transactions and contracts and answer queries from other nodes – light nodes, or not yet synchronized full nodes – about the current state of the blockchain.

Light nodes, as the name indicates, are less memory and computationally demanding. They do not verify every block and usually do not possess a copy of the current blockchain state and therefore rely on full nodes to provide the missing information.

There is also a third, special kind of peers – bootstrap nodes. These are highly available nodes run by volunteers whose purpose is to facilitate the connection of new clients by providing them "enode identifiers" of other nodes in the network. The enode datatype is represented by a unique hexadecimal identifier concatenated with an IP address and a TCP port.

Client Implementations

There are currently eight implementations of Ethereum clients written in eight different languages. All the implementations use the same set of API calls, specified by the WIRE[10] protocol, allowing querying about the information stored on blockchain, sending transactions and so on.

[9] https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/
[10] https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol

Their logic, synchronizing strategies or the way of storing the data describing the state of the platform can substantially differ.

In this work, we decided to use the Geth implementation for our experiments. This implementation is considered to be stable, reliable, and is currently, by far, the most used one [KMM+18].

This open-source client implementation of the Ethereum protocol provides a wide range of use-cases. It allows acting as a miner, a bootstrap node or to control an external account with the possibility to send transactions, deploy contracts and so on. Geth runs on multiple platforms and can be controlled either by a built-in interactive JavaScript console or by API calls.

2.5 Information Propagation

The efficiency of the Ethereum platform depends on the time it takes for a transaction to propagate from users to miners and the time it takes for a transaction to get committed.

In the following paragraphs, we describe the approach used for the propagation of blocks and transactions. We will also discuss transaction committing.

2.5.1 Transaction Lifecycle

When an external account decides to create a new transaction, it broadcasts the transaction to all neighboring peers it knows about. On average, the Geth implementation maintains connections with 25 other nodes, whereas Parity connects to 50 peers. Upon a transaction reception, the nodes forward the transaction to their neighbors – usually just to a limited number of them, but that is implementation dependent – therefore the transaction propagates through the network until it reaches a miner. According to [KMM+18] Geth propagates transactions to all its neighbors and Parity forwards transactions just to square root of its neighbors, i.e. to 25 and 5 peers respectively. A miner includes the transaction into a new block only if: its parameters are valid, the funds are sufficient, the signatures match and the miner finds the offered gas to be profitable for them.

A transaction is considered to be committed after it appears in a block included in the main chain, with at least 12 successors. It is important to mention that transactions may never commit. Firstly, miners can for any reason refuse to include a transaction in a block. Secondly, even after getting included in a block, that block may not become part of the main blockchain. In both cases, the user may still try to broadcast the transaction again. One can be sure that a transaction will never commit when another transaction from the same account with the same nonce gets committed.

2.5.2 Block Propagation

Wire protocol – also known as the Ethereum subprotocol – defines three ways of block propagation:

- Several whole blocks can be propagated in chunks. This method is used only when the peer is synchronizing with the network and requests a specific set of blocks.
- Individual blocks can be propagated as a whole (header + body). This is the most common way of block propagation.
- Unique hash of a block can be propagated separately without the burden of block's body. The peers that receive this announcement, request the whole block data only if they do not possess the attached block yet. The advantage of this method lies in faster propagation due to smaller size of message.

After block reception, the peers perform various verifications such as whether the block's nonce leads into the particular mixHash. Then, they verify and execute individual transactions and update so the state of Ethereum locally.

With the exception of the first type of message, peers upon new block reception forward the message to their neighbors.

Chapter 3

Related Work

This chapter discussing the work of others is composed of four parts. It starts with Section 3.1 which gives a brief introduction to the problematics of Bitcoin.

The succeeding Section 3.2 covers the imperfections of blockchain systems, such as scalability issues, common design flaws or some specific pitfalls – e.g. the selfish mining attack or the blockchain anomaly.

Then the Section 3.3 deals with the network plane of blockchain systems. The referenced papers investigate properties like the propagation time of transactions and blocks or the state of Ethereum's network decentralization. In addition, they deal with the causes of delayed transactions and the discovery techniques of influential nodes amongst others.

The last Section 3.4 briefly discusses papers concerning security threats to Ethereum smart contracts.

3.1 Bitcoin

Bitcoin [Nak08] represents a revolutionary electronic cash system which eliminated the need for a trusted third party to prevent double-spending by pioneering the technique of hash-based PoW chain of records; also known as "blockchain". Bitcoin started the world's interest in decentralized cryptocurrencies, which later evolved into even more sophisticated decentralized platforms like Ethereum. We have already covered the inventions proposed in the original Bitcoin paper in Section 2, so we will move straight away to discuss Bitcoin's weaknesses which appeared after publishing the white paper.

Bitcoin suffers from a wide range of problems [ES14, GBE+18, CDE+16]. Cryptographers are worried about the security of the whole system since the accounts are represented by public keys based on elliptic curve cryptography, which – as other popular public key cryptosystems

like RSA – has no mathematical proof of difficulty. Meaning that even though the encryption algorithm appears to be, and is secure against brute-force and even more complex attacks. Still, there is no proof that there is a way to effectively compute the corresponding private key, and therefore be able to sign transactions of funds of other people and break the whole system. In addition to that, there already are quantum algorithms that destroy the security of elliptic curve cryptography. Luckily for Bitcoin, there are no quantum computers with the sufficient number of qubits yet. Although, the current trends show that they may be invented in the near future. Therefore, a better idea would be using some cryptography algorithm resistant to both threats, e.g. the novel Trans-Vernam cypher BitFlip [SP17], which offers secrecy close to the Vernam's perfect secrecy but with more convenient memory requirements.

Bitcoin, as well as any other platform based on the Nakamoto's consensus algorithm, only performs well with a limited number of users and a limited quantity of transactions. Beyond a certain limit, for instance, if Bitcoin had to deal with the same amount of transactions as VISA does; it would collapse as the system does not scale. In order to solve this problem, it would be needed to completely redesign the whole system, as discussed in [CDE+16]. Bitcoin also suffers from continuously growing blockchain, which may easily exceed 200 GB before the end of 2018.

Bitcoin is dealing with serious economical issues; such as the inherently unstable exchange rate caused by the nature of decentralized currencies that miss a central authority – e.g. a central bank – to control the money supply. In addition, there currently are no reliable cryptocurrency exchanges, the transaction fees are high and the legal issues are not resolved yet.

Bitcoin and its PoW mechanism causes an environmental threat since it already consumes more electricity annually than Portugal (57.8 vs. 46 TWh/year in April 2019) and the consumption steadily grows.[1] However, those non-technical problems are out of the scope of this document.

In the following text, we present the technical issues of the blockchain technology.

3.2 Blockchain Shortcomings

3.2.1 Scalability

The authors of [CDE+16] discuss a very actual and severe problem which lies in the fact that current blockchain systems do not scale and thus are not prepared for mass usage. The final outcome of this paper is that the scalability issues cannot be resolved just by reparametrization; instead, a fundamental protocol redesign is needed. The authors examine and perform measure-

[1] https://digiconomist.net/bitcoin-energy-consumption

ments in the Bitcoin network, however, the work outcomes apply to all blockchain based systems that follow the Nakamoto's consensus algorithm and its derivates.

The authors identified that Bitcoin's maximum possible throughput is 3.7 transactions/sec, which had simply been calculated from the maximum block size and the inter-block time. Time of transaction confirmation is between 10 to 60 minutes. The time needed to download and process the history necessary to validate the current system state is roughly four days. And finally, the costs of resources of the entire system for a single transaction, which include the costs of mining, storage or transaction validation were estimated to be between 1.4 - 6.2 American dollars[2]. Hence, all those values make Bitcoin not usable as a global payment system.

Furthermore, the authors were verifying whether it would be possible to improve scalability simply by reparametrization, i.e. by changing the maximal block size, the inter-block time and other values, while maintaining the current degree of decentralization. This means that at least 90 % of nodes must be able to correctly participate in the network – be able to receive and verify blocks in time. They came up with the idea to improve the system latency by reducing block intervals. Their measurements showed that to retain the 90 % effective throughput and fully utilize the bandwidth of the network, the inter-block time should not be lesser than 12 seconds. That is because 12 seconds represents the time it takes to propagate a 80 KB block to 90 % of nodes. This size of block was chosen because their measurements had revealed that in blocks greater than 80 KB[3], the block size represents the dominating factor of prolonged propagation times; whereas in blocks smaller than this threshold, the network latency represents the bottleneck.

Overall, the authors claim that tuning by reparametrization is possible but still cannot help Bitcoin to be prepared for mass usage. Therefore, in the rest of the paper, they discuss techniques that would allow blockchains to scale better.

They performed measurements showing that Bitcoin's network protocol does not fully utilize the network bandwidth. That is because Bitcoin nodes, before propagating a transaction, must perform a verification which delays the whole transaction disseminating process. The second bottleneck lies in the fact that every transaction is sent twice. First on its own, and then as a part of a block. Another problem is that peers are not incentivized to propagate transactions. To fix the first two problems, it is needed to make significant changes to the protocol. Their idea on how to motivate nodes to propagate transactions is to offer a small fee for doing so. The authors also mention that Bitcoin may benefit from an improved consensus protocol, namely they

[2]The estimations were calculated in years 2015 and 2016. At the end of 2017, the fees were as high as 52 dollars caused by enormous demand for Bitcoin by speculators.

[3]As average connectivity speeds steadily grow, and latencies decrease, we expect the current block size threshold to be greater than 80 KB, as it is written in the cited work from 2016.

mention the GHOST protocol, which significantly increases consensus speed and bandwidth while sustaining the level of security.

They mention one technique which could really fix the scalability problems in the long term. It is called "sharding" and consists of splitting up the tasks of consensus among sets of nodes. Unfortunately, to achieve that without breaking other properties of the system is much harder than it sounds. Some blockchain systems, for instance Ethereum, are actively researching these problematics and are planning implementation in the next years. Their authors also propose some more techniques but to keep report brief and concise; we will end here.

To conclude, the authors provided very interesting measurement results and calculations about how to reparametrize Bitcoin system in order to help to scale better in the short term. They also discussed various long-term approaches to the possibility of resolving the scalability issues of blockchain systems in a definite way. However, they surprisingly omitted the very interesting approach of building a consensus system based on the novel directed acyclic graph data structure called Tangle [Pop16] used by the cryptocurrency IOTA[4]. Tangle based systems do not suffer from scalability issues and do not even require transaction fees. On the other hand, it is not certain whether Tangle indeed allows for a purely decentralized platform, since IOTA currently relies on a centralized mechanism to assure security.

3.2.2 Double Spending

Another paper [NG16] identifies the conditions under which blockchain systems fail to ensure consensus and so prevent someone from executing dependent transactions, such as “Bob transfers some coins to Carole only if he received coins from Alice”. The authors also show in an experiment on a private Ethereum network that this anomaly may lead to the double spending attack, although it requires the attacker to be able to mine new blocks. Therefore, it is not feasible in the Ethereum’s main network, unless the attacker controls substantial mining power. They propose a solution which consists of using special smart contracts instead of sending simple transactions as usual. But that may be too complicated for average users. Therefore, they came up with recommendations about how to improve the Ethereum protocol, for instance by adding logical clocks to mark causally-related events.

3.2.3 Selfish Mining

The authors of the highly cited paper called "Majority is not Enough: Bitcoin Mining is vulnerable" [ES14] from 2013 show that the Bitcoin protocol is vulnerable to the "Selfish-mining"

[4]https://iota.org

technique, which presents a way how to perform more profitable mining by acting dishonestly by not immediately propagating newly mined blocks. The idea is the following: if a dishonest miner postpones the release of a new block and manages to mine the subsequent block fast enough, the other competing miners would waste their resources by mining on an outdated block. This leads to higher reward for the cheating miner.

The authors performed an extensive mathematical analysis of the Bitcoin network with a selfish miner using Monte Carlo simulation instead of solving a real cryptopuzzle. They simulated the behavior of the platform with one selfish miner – with variable mining power – and the rest of the network acting honestly. The results show that selfish-mining is usually profitable already on pools with the computational power higher or equal to one quarter of the overall network mining power. Under certain circumstances, the technique can be effective on a pool controlling just 10 % of the total power. The authors point out a consequential problem; the rational miners would like to join the dishonest pool in order to increase their profits and therefore a selfish pool may rapidly grow and became soon the only mining entity and so take control over the Bitcoin protocol.

The authors propose an improvement to the protocol which may significantly mitigate the problem by forcing the miners learning of competing branches of the same length, to propagate all of them and choose the one to mine on randomly. Overall, this paper revealed a very serious vulnerability. It has been six years since publishing and the vulnerability is still actual and threatening.

In [NF19] the authors develop an analysis for selfish mining in Ethereum. They accomplish that by following the strategy invented in [ES14] for Bitcoin, with certain changes because of the GHOST protocol that is present only in Ethereum and which has significant impact on selfish mining.

The Ethereum's uncle rewards (from the GHOST protocol) were initially meant to mitigate mining centralization by compensating miners with lesser computational power. As it shows, however, these rewards can reduce the cost of launching selfish mining. That is because the uncle blocks mined by the selfish pool can always be referenced with block distance one, which is worth the maximum uncle reward (7/8 of the base block reward), whilst the honest miners' uncle blocks can only obtain rewards for blocks with greater distance. These are worth between 1/8 and 6/8 of the base block reward. The authors propose a solution to this problem. They recommend to decrease the reward for immediate neighboring uncle blocks and increase the reward for the uncles with longer distances. In their work, they calculate the exact hashing power threshold of making selfish mining profitable and provide with mathematical proofs of their claims.

We suggest to take [NF19] very seriously because Ethermine, the currently dominant mining pool, controls more than one quarter of the computational power of the network. That is – according to the outcomes of the work – enough to perform the selfish mining attack.

3.2.4 Design Flaws

The authors of [CV17] investigate and discuss the blockchain systems' design flaws with the aim at the "permissioned" systems; in the sense that their participants are identified. They show on specific examples how faulty protocols are being used, for instance by mentioning an enterprise-ready, smart contract platform that follows the QuorumChain protocol, which as they show, cannot ensure consensus. Another example represents the IOTA Tangle which is a novel and scalable blockchain alternative. They correctly point out that the Tangle developers just claim that it ensures a similar level of consistency as the blockchain systems. However, no formal proof or analysis is available. Despite that, people do not hesitate to invest in this technology; currently[5] all IOTA coins in the circulation are worth about 832 million US dollars.

To conclude, the authors of this paper discuss serious problems which everyone involved in blockchain technologies should be aware of. The outcome, and main idea of this paper, is represented by the authors' advice which aims for blockchain system designers to follow the established practice from cryptography relying on formal proofs, detailed models and public reviews.

3.3 Blockchain Systems' Network Plane

Weber et al. [WGP+] investigate the Bitcoin and Ethereum networks with the aim of measuring network metrics, such as the transaction commit times or the prevalence of uncle blocks. In order to perform the measurements, the authors modified the open-source implementations of Bitcoin and Ethereum clients to gather all incoming transactions and block announcements. In addition, they changed the client configurations to allow connecting to a big amount of other peers and performed a few other minor changes to the client implementations.

They came up with several interesting observations. They have found that network reordering, gas price and gas limit affect commit times. Their measurements also showed that there is no correlation between maximum gas property and commit delay. Another observation is that the higher the gas price was, the less likely a long delay was observed (up to 25 Gwei). They point out the problematic issue that some transactions can never commit. Another outcome of

[5] In 30 April 2019.

their work is the idea for implementing a mechanism to explicitly abort "stuck" transactions even though there already are some "intricate" ways to achieve that.

The paper is very important for our work since we will base our measurement techniques on theirs. Nevertheless, in our opinion, they have omitted measuring one – for a P2P system – crucial property, which is the propagation time of transactions and blocks between geographically distant nodes.

This property is measured in a paper called "Information Propagation in the Bitcoin Network" [DW13]. The authors performed the measurements as follows: first, they deployed a slightly modified Bitcoin client, which was connected to a large sample of other peers, and let it listen for block and transaction announcements. For every event, they accordingly recorded the local time to allow measuring the propagation times of blocks and transactions.

Moreover, they performed a second experiment where they deployed a node which interconnected all reachable nodes in the network. Doing so caused a substantial decrease of propagation times and it showed that the most influential cause of propagation delays is the distance between the origin of a transaction and the nodes. The interconnection of nodes also caused more than a two-fold decrease fork rate; therefore they concluded that the propagation delay is the primary cause for blockchain forks. They proposed various suggestions about how to speed up the propagations, such as immediately forwarding of the "INV" messages announcing new transactions, or dividing the block verification into two phases – which would allow relaying the blocks as soon as the difficulty checks are done. But those, as the authors confess, are just short-term solutions.

3.3.1 Bitcoin's Topology

The authors of "Discovering Bitcoin's Public Topology and Influential Nodes" [MLP+15], as the name suggests, analyze the topology of the Bitcoin network. Moreover, they try to reveal gateways of mining pools and generally investigate how the nodes are connected. They are interested in the node interconnection because it affects which nodes have advantages in the faster propagation of data; or contrarily, bad interconnection may cause some attacks feasible to perform.

They developed the CoinScope infrastructure which enables maintaining long-lived connections and supports for extensive short-term scans of the entire network. They invented the AddressProbe technique for P2P links discovery in the Bitcoin network. In order to perform the tests, they deployed five Bitcoin nodes connected to the main network. Each of them, every two minutes performed short bursts of "GETADDR" requests to collect a list of all active connections and created a snapshot of the network. Their technique cannot observe NAT'ed clients and those

which do not wish to be observed; nonetheless, it provides very reliable results.

In addition, the authors introduced a method for finding influential nodes called decloaking. This method exploits the fact that clients, upon transaction reception, add the transaction to its pool of incoming transactions only if it does not conflict with any other transaction from the pool. If there is a conflict, then they accept the first received transaction. Therefore, "coloring" each node with a different conflicting transaction and repetitively sending those transactions to all the nodes would lead to revealing the most influential nodes since their transactions would appear in blocks more often. Sending distinct transactions to every node would be impossible to do simultaneously because of varying network latencies and unequal quality of connections between nodes. Nevertheless, the authors came up with a feasible technique, exploiting the fact that the Bitcoin protocol does not simply broadcasts new transactions to all neighbors, but instead follows a three-round protocol. The first step is to broadcast an "INV" message with a hash of the transaction and wait for eventual "GETDATA" message requesting the particular message. A reception of subsequent "INV" messages with the same hash times out for two minutes any outgoing "GETDATA" messages providing a time-frame, in which selected transactions can be sent to particular nodes without having to win a network latency race; therefore allowing for finding influential nodes.

The results show that about 2 % of nodes, which very probably are gateways to mining pools, represent three-quarters of the total mining power. Furthermore, the authors claim that the topology of the Bitcoin network is not purely random. They hypothesize that it is caused by two reasons – firstly, because nodes use a small set of DNS nodes to connect to the network, and secondly because of the slow learning about geographically distant peers.

3.3.2 Decentralization in Bitcoin and Ethereum

The authors of the paper "Decentralization in Bitcoin and Ethereum Networks" [GBE+18] from January 2018 investigate the decentralization of both networks and their robustness against attacks. They also perform various measurements about network properties of nodes or the interconnection among them.

They deployed the Blockchain Measurement System (BMS) consisting of multiple observing nodes, which sequentially collect data from other peers on the network. A peer's bandwidth is measured by requesting a large amount of data by each BMS node and capturing the flow – note that the observing BMS nodes run at powerful servers with 1 Gbps links so one can be assured that they do not represent the bottleneck. The measurements are repeated over time to give more precise measurements. The results show that the median bandwidth for a Bitcoin

node has increased by a factor of 1.7x in just one year. The overall results also show that the average Bitcoin node throughput is 56.1 Mbps, whereas an average Ethereum peer possesses just a 29.4 Mbps fast network up-link. The measured results appear to be very precise, as the authors performed tests on clients controlled by them to confirm the accuracy of the BMS; the measured bandwidth consumption differed just by 0.2 Mbps from the real values.

Since it is not possible to measure P2P latency between nodes which are not under our control, the authors decided to measure the delays of ICMP pings from BMS nodes to all peers on the network and then established bounds from observed latencies by following the method invented in [FJJ+01]. They also measure the approximate geographical distance between peers but the techniques used to perform this are out of the scope of our work. Nevertheless, the results show that Bitcoin has many more geographically close nodes and the latency also plays in favor of Bitcoin whose median latency is about 109 ms, whereas the median message delay in the Ethereum network is 152 ms. At first sight, the results look very positive for Bitcoin. Nevertheless, the authors speculate that the geographical proximity and low latency in Bitcoin are caused by the fact that many of its nodes are run in data centers. The geographical results show that Ethereum nodes are much more evenly distributed over the network.

The authors measured the distribution of mining power by comparing the ratios of main chain blocks generated by distinct mining entities. Then, they manually processed identities of miners to detect and merge duplicates, such as pools operated by the same administrator. The results show that Ethereum has slightly more centralized mining, where on average 61 % of the weekly power was shared by only three miners in contrast of 53 % mining power represented by the four strongest miners in Bitcoin network. The mining power of the strongest miner in the Ethereum network ranged from 21 to 26 % of the overall power whereas in Bitcoin the percentage never surpassed 21 %. Those numbers are alarmingly large but still acceptable. Another observation of their measurements is that Ethereum has lower mining power utilization (around 92 %) than Bitcoin (99 %); obviously caused by Ethereum's higher block generation frequency.

3.3.3 Ethereum Network Properties

The authors of the recent paper called "Measuring Ethereum Network Peers" [KMM+18] investigate the underlying network of Ethereum and present interesting observations that are very relevant to our work. They develop NodeFinder, which is a modified Geth node that aims to observe nearly all DEVp2p peers and capture particular network metrics.

During three-month lasting measurements NodeFinder managed to observe and connect to more than 15 thousand Ethereum peers, which is around 2.3 times more than prior work [GBE+18].

This is because NodeFinder ignores max peer limits at both the DEVp2p and Ethereum layers, and stubbornly searches the whole network for new peers. Moreover, it disconnects from peers after a successful Ethereum handshake whilst a normal Geth client maintains connections until either side requests disconnection or due to some failure. This saves tremendous amount of traffic generated by Ethereum connections as they show in a case study. For the needs of the measurements, they save addresses of previously seen peers and periodically reconnect to them for short time only.

In contrast with other cryptocurrencies such as Bitcoin or Monero; Ethereum does not run on an isolated network. Instead it operates on the Ethereum subprotocol that makes part of the multi-platform network DEVp2p. The measurements reveal that 48.2 % of DEVp2p nodes are useless for the Ethereum platform, meaning that they either do not run the Ethereum subprotocol, or they do not operate the main Ethereum blockchain. Around 6 % of DEVp2p nodes are Non-Ethereum services, such as Swarm decentralized storage service, the Whisper communication protocol, or various light client protocols. Moreover, within the Ethereum subprotocol, individual peers can operate on various subnetworks. In addition to the main network, there are various Ethereum test networks operating; the two biggest Ropsten and Rinkeby together account for more than 9 % of the total number of peers. Finally other cryptocurrencies, such as Musicoin (1.5%), Pirl (1.5%) and other also operate on DEVp2p.

.

As a consequence of the hard fork that happened on 20 July 2016, two separate cryptocurrencies Ethereum and Ethereum Classic run next to each other on the same subprotocol, with the same genesis block hash and with the same network number; causing significant network noise. According to the results of measurements, they classified 97 074 peers definitely as non-Classic and 3 386 as Classic. This is a problem because current implementations of Ethereum peers maintain connections with these useless peers and waste so their resources.

The authors discovered that 98.77 % of all peers run three different implementations only, namely: Geth (76.58 %), Parity (17.04%) and ethereumjs-devp2p (5.15 %). In most of cases, the clients are regularly updated to the newest version. But the authors also point out that 3.5 % of the Geth nodes run version so old that they are incompatible with the rest of the network.

The authors took a closer at the two prominent implementations Geth and Parity and discovered a bug in the latter. Both clients use a log distance metric for calculating a position to for the new peer's inclusion in a table. There is logical bug in Parity's implementation which, although it does not break compatibility between Parity and other clients, it likely impedes the RLPx node discovery and in the worst case it may lead into an unintentional eclipse attack, i.e.

obscuring a node's view of the rest of the platform.

The study performed in [KMM+18] also informs about the geographic and network distribution of Ethereum peers. The results say that 43.2 % of nodes operate in he US and 12.9 % in China. The authors also discovered that just 8 big cloud providers, such as Amazon, Alibaba, or Google, account for 44.8 % of nodes. In addition, they had measured peer latencies using the techniques explained in [GBE+18, FJJ+01] and discovered that Ethereum has higher average latencies between peers compared to Bitcoin (145 vs 173 ms) indicating that Ethereum is formed in a more random manner.

In the end the authors offer several suggestion about how to improve the Ethereum project. They advocate for automatic updates of clients since vulnerabilities already resulted in financial loses in the past. Next, they claim RLPx, DEVp2p, and Ethereum to be insufficiently documented and hypothesize that this contributed to conflicting client implementations that eventually lead into the possibility of eclipse attacks.

3.4 Smart Contracts

Correct execution and resistance against attacks represent crucial properties for smart contracts. A successful attack might lead to the stealing of funds which is undesirable for everybody. Therefore, the authors of [ABC17] analyze the security vulnerabilities of Ethereum smart contracts and extensively examine these problematics. They provide a taxonomy of the causes of vulnerabilities and show an example attack for each of them. The authors observed that the common cause of insecurity of smart contracts is the difficulty to detect the difference between their intended behavior and the actual one. They conclude that verification tools may help but the choice of the Turing-complete language limits the possibility of verification. To conclude, this paper provides a very extensive list of attacks where they covered all the major vulnerabilities and attacks reported so far. Therefore, we find this paper as a very useful for every smart contract developer.

Another paper discussing the security of smart contracts is called "Making Smart Contracts Smarter" [LCO+16], where the authors investigate the security of running smart contracts on the Ethereum network and document several new classes of security bugs. They also present the Oyente tool for finding potential security threats. The tool follows a novel symbolic execution approach. Basically, the Oyente takes both the bytecode of a contract and the current Ethereum global state, then it performs test executions and various bug detections. Afterwards, it produces a list of potential vulnerabilities. The authors ran Oyente on the real Ethereum network. Their most exciting result was that it marked 8 833 out of 19 366 existing Ethereum contracts as vulnerable. The tool, as the authors prove, has a false-positive rate smaller than 7 %.

In the paper called "A concurrent perspective on Smart Contracts" [SH17], the authors took a very original approach to look at smart contracts. They compare accounts using smart contracts in a blockchain to threads using concurrent objects in shared memory. This leads to the possibility of reusing existing techniques, tools, etc, from the distributed programming area. They point out that even though the smart contracts are executed sequentially, a contract logic may communicate with the world outside blockchain; enabling concurrent behavior that can be exploited by adversarial parties. They show on two real-world examples how that was exploited – for instance, on the example of the famous DAO bug.

The very last paper [DGHK17] that we cover in this section elaborates on the execution efficiency of smart contracts. Since Ethereum smart contracts do not allow for concurrent execution and therefore do not fully utilize the potential of current multi-core processors; the authors thought that permitting parallel execution of smart contracts may not be a bad idea. Their approach for that is the following: miners perform a parallel execution of transactions adapting techniques from software transactional memory systems – so in case of a reached conflict, they simply roll back and perform the execution of conflicting transactions sequentially. At the same time, they discover a schedule of parallel execution for the validators.

The authors performed an experimental evaluation using a prototype implemented in the Java Virtual Machine since the EVM does not support multithreading. Their results showed that adding concurrency to smart contracts is possible and it really improves their effectivity. The prototyped implementation demonstrated a 1.33x speed-up for miners and 1.69x for validators on a setup with three threads. Unfortunately, the authors have not explained what causes such overhead nor whether it would be possible to reach a better speed-up.

Chapter 4

Approach

In this chapter, we start by presenting the description of all metrics from this work. Then, in Section 4.2, we show the high-level approach of measuring for each single metric. Section 4.3 describes the two types of measurements performed in this work. Finally, in Section 4.4, we comment how the data gathered on our infrastructure is processed and how the final metrics are calculated.

4.1 List of Metrics

All in this section listed metrics are captured on our modified version of the open-source Geth client that is deployed on an infrastructure consisting of four geographically dispersed instances. The infrastructure, as well as the modifications to the client, are thoroughly described in Chapter 5.

- **Propagation time of blocks:** The time it takes for a block to propagate from a miner to various geographically distributed nodes.
- **Propagation time of transactions:** The time it takes for a transaction to propagate between geographically dispersed nodes.
- **Transaction commit time:** The time it takes from the first observation of a particular transaction to the time of the transaction to be included in the main blockchain with twelve succeeding blocks. When a block is followed by this amount of successors, it is practically impossible to revert that block due to the need of immense computational power. Therefore, we call transactions included in such blocks as committed.
- **Transaction reordering:** This metric compares commit times of "in-order" and "out-of-order" transactions. The sender stamps every transaction with a nonce, which is incremented by 1 for each new transaction. We say that two transactions assembled by the

same sender were received out of order, when we first capture the transaction with higher nonce.

- **Gas usage per transaction type:** We distinguish between three transaction types:

  - regular value transfers
  - contract creations
  - contract function calls

  In this metric, we measure for each type, firstly, the number of captured transactions and secondly, the average amount of used gas.

- **Impact of gas price:** The correlation between the commit time and the gas price offered for including given transaction.

- **Never-committing transactions:** The percentage of captured valid transactions which never became part of any block on the main blockchain.

- **Prevalence of forks:** This metric measures the total number of blockchain forks, where we distinguish between forks of lengths two, three and possibly longer ones. We also calculate the proportion of observed forked blocks and the total number of blocks.

- **Transaction reordering in forked blocks:** The degree of reordering of transactions included in forked blocks and associated blocks on the main chain.

- **Duplicate transactions:** Duplicates are those transactions included in some block on the main chain and at least in one uncle block. This metric captures their proportion to the total number of transactions on the main chain.

- **Block reception redundancy:** For each unique received block announcement, we measure the number of redundant receptions of that particular block.

- **Invalid Network Messages:** The Ethereum network peers communicate using messages to transfer all possible data ranging from block header announcements, block bodies, data requests and status messages to individual transactions. All messages undergo a validation phase verifying proper following of the protocol, e.g. valid formatting or maximal size compliance.

  An increased volume of such invalid messages may indicate a possible network attack. For this reason, in this metric, we measure the number of invalid messages per message type.

- **Empty Blocks:** The number of blocks that do not contain any transaction.

4.2 Measurement Techniques

In this section, we present the techniques of measuring for each single metric.

4.2.1 Propagation Time of Blocks

There is no standardized way about how to capture the exact time it takes for a block to propagate from a miner to various geographically distributed nodes. Therefore, below we present four alternative techniques that we consider, and discuss which is the best for our purposes.

Querying Mining Pools' API

Since the vast majority of blocks is produced by a small set of mining pools and all these pools provide an API offering basic information about the blocks; the most straightforward technique would be simply querying the pools for appropriate data – i.e. the time of block dissemination – and compare the captured times at which each block was received at our measuring nodes. The great advantage of this strategy is that it would enable us to determine the time between the moment when the block was first disseminated and when the block was received.

The main problem of this technique lies in the fact that we have no possibility to synchronize the clocks of our measuring nodes with the clocks of miners. Therefore, the results would be unreliable.

Measuring on Mining Pools' Gateways

Every prominent mining pool propagates its newly mined blocks by a set of gateways which, under the hood, are classical Ethereum peers. Therefore, if we managed to locate these nodes and connect to them directly, we would be able to precisely capture the time of dissemination. The problem is that IPs of the pools' gateways are not publicly known and locating these nodes is very difficult, as the authors of [MLP+15] have shown.

Querying a Mining Software

Another possible approach is connecting to every prominent mining pool as a regular miner using a certain mining software – e.g. Ethminer[1] or Mist[2]. Then, it would be possible to capture the time of block dissemination by listening for a message from a particular command and control (C&C) server sending us a set of data needed to mine a new block. A C&C server sends this message when it finds a new block or when it learns about a newly mined block from another miner.

This technique looks promising, yet has one disadvantage. Even though we would be able to cover a vast majority of blocks, still we could not capture them all. This is because we would

[1] https://github.com/ethereum-mining/ethminer
[2] https://github.com/ethereum/mist/

miss the blocks generated by the individual miners that do not participate in any mining pool.

Monitoring the Ethereum Network

This technique was originally proposed in [DW13] where the block propagation time is measured using an infrastructure consisting of a set of geographically distributed nodes with synchronized clocks listening for new block announcements. The time of the first captured announcement on some of the nodes is recorded and later compared with the times it had taken until other nodes received the same announcement.

By using this technique we would cover every single block in the network and not just those mined by prominent pools. Another advantage is that we could directly compare our results for Ethereum network with the results from [DW13] where the authors measured the Bitcoin network using exactly the same technique. This lets us do a comparative study of both platforms. And finally, this method is not very difficult to perform.

On the other hand, there is the threat that if each of our monitoring nodes was geographically far from the miner, the measured delay would be substantially smaller than the real one. We are, however, more interested in observing the correlation between the geographical distance of nodes and the propagation delays, which this method captures precisely, than the exact time it takes to propagate from a miner to nodes. For this very reason, we decided to use this method in our work.

4.2.2 Propagation Time of Transactions

We follow the same technique measuring propagation delays presented in Section 4.2.1. In this case, however, we are interested in messages carrying transactions instead of blocks. We deploy a set of Ethereum nodes listening for transaction announcements. Afterwards, we compare the time of the announcement reception on the node which had observed it first, with the times that the other nodes needed to receive that same announcement.

4.2.3 Transaction Commit Time

In this metric, we replicate the measurements methodology from [WGP+]. The idea is to capture the local time upon a new transaction observation and then measure the time it takes for the transaction to become part of some block included in the main blockchain with twelve succeeding blocks.

4.2.4 Transaction Reordering

Every transaction produced by the same account must be executed in a particular deterministic order. For this reason, the transactions are stamped by the sender with a nonce which is incremented by 1 for each new sending. When a miner receives a transaction with a nonce $n + 1$, they cannot include it into a block until they obtain the n-th nonce. Such occasions are called "out-of-order" transaction receptions. We speculate that "out-of-order" receptions lead into an increased time needed to commit since their possible inclusion in a block is delayed by waiting for their predecessors.

This metric compares the commit times of "out-of-order" and "in-order" transaction receptions with the aim to confirm our hypothesis.

4.2.5 Gas Used per Transaction Type

Transactions can either serve for money transfers, contract creations or function invocations. In this measurement, we capture the number of their occurrences. In addition to that, we measure the amount of used gas for each transaction type.

4.2.6 Impact of Gas Price

In this metric, we analyze the correlation between the commit time and the gas price of a transaction which is defined by the sender. It is expected that high gas price will lead to a higher probability of low commit time. This happens because miners are incentivized to include those transactions to make more profit. This metric is in order to observe whether it is possible to speed up the commit time by offering a higher gas price.

4.2.7 Never-Committing Transactions

It may happen that transactions can never commit – as we have already mentioned in Chapter 2. We can count the exact number of transactions that will certainly never commit by observing already committed transactions with the same nonce which originated from the same account. Additionally, we measure the transactions that were captured by our peers but were never included in a block.

4.2.8 Prevalence of Forks

This measurement provides the number of:

- blocks included in the blockchain

- forked blocks later referenced as uncle blocks
- unrecognized forked blocks – those forks not referenced by any block from the blockchain

Since our observation nodes should capture every block announcement on the network, the resulting numbers should be extremely precise. We also capture the total number of observed forks, where we distinguish between forks of various lengths. Finally, we compare whether the fork's length somehow correlates with the block's probability of becoming recognized by any block from the blockchain.

4.2.9 Transaction Reordering in Forked Blocks

This metric provides the degree of transaction reordering between the transactions included in forked blocks and the corresponding blocks on the main chain.

4.2.10 Duplicate Transactions

The transactions which are included both in an uncle block and in a block on the main chain are called duplicates. In this measurement, we identify the proportion of unique transactions and duplicates.

4.2.11 Block Announcement Reception Redundancy

This metric measures the number of times a unique block is received; thereby assessing block reception redundancy. The results interest us because they will reveal the network traffic overhead caused by the redundancy of data receptions.

4.2.12 Invalid Network Messages

All the received network messages that do not pass through validation are captured in this metric along with the type of message. The information provided by this metric may be useful for identifying network attacks.

4.2.13 Empty Blocks

Miners are incentivized to include as many transactions as possible in blocks in order to earn Ether for the fees that each transaction is worth. Some miners, however, intentionally mine blocks without any transactions to save time with their validation, and to make the blocks smaller for faster propagation. By doing this they increase their chances to be the first to propagate their blocks.

In this metric, we measure the number of empty blocks and their proportion to all blocks. In addition, we investigate whether empty blocks indeed become main blocks more often. We are interested in this metric because an increased number of this type of blocks is very harmful for the Ethereum platform as it causes prolonged transaction inclusion times.

4.3 Measurements

4.3.1 Main Multiple-node Measurement

We performed one-month lasting measurements on all four machines from April 1st to April 30th. This timeframe provides enough data to reliably capture the properties of the Ethereum mainnet.

4.3.2 Secondary Single-node Measurement

In the metric measuring block announcement reception redundancy we are interested in knowing how many block a regular Geth client with default settings receives. Our infrastructure, however, runs clients that are connected to a significantly greater number of peers than is the default number. For this reason we perform a secondary measurement that runs on a single machine only. This instance runs with the default number of peers, which is 25 and the measurement takes place from May 2nd to May 9th.

4.4 Logs Processing

Figure 4.1 illustrates the process of gathering and processing the Geth-generated logs later used for calculating metrics. The diagram is divided into 16 steps that are explained in the following text.

The green column (Step 1) represents the software running on each machine; that is the Geth client and Rsyslog – both already introduced in the previous Section 5.2.

At the end of the measurements, there are six log-files that are shown in the grey column (Step 2). The four log files colored in green are final whilst the yellow ones must be processed, which is done in Steps 3 - 16, before they could be used for calculating metrics. This is achieved by the usage of Bash and Python 3 scripts and the Pandas[3] library. We describe each of the six log files in separate subsections below.

[3]https://pandas.pydata.org/

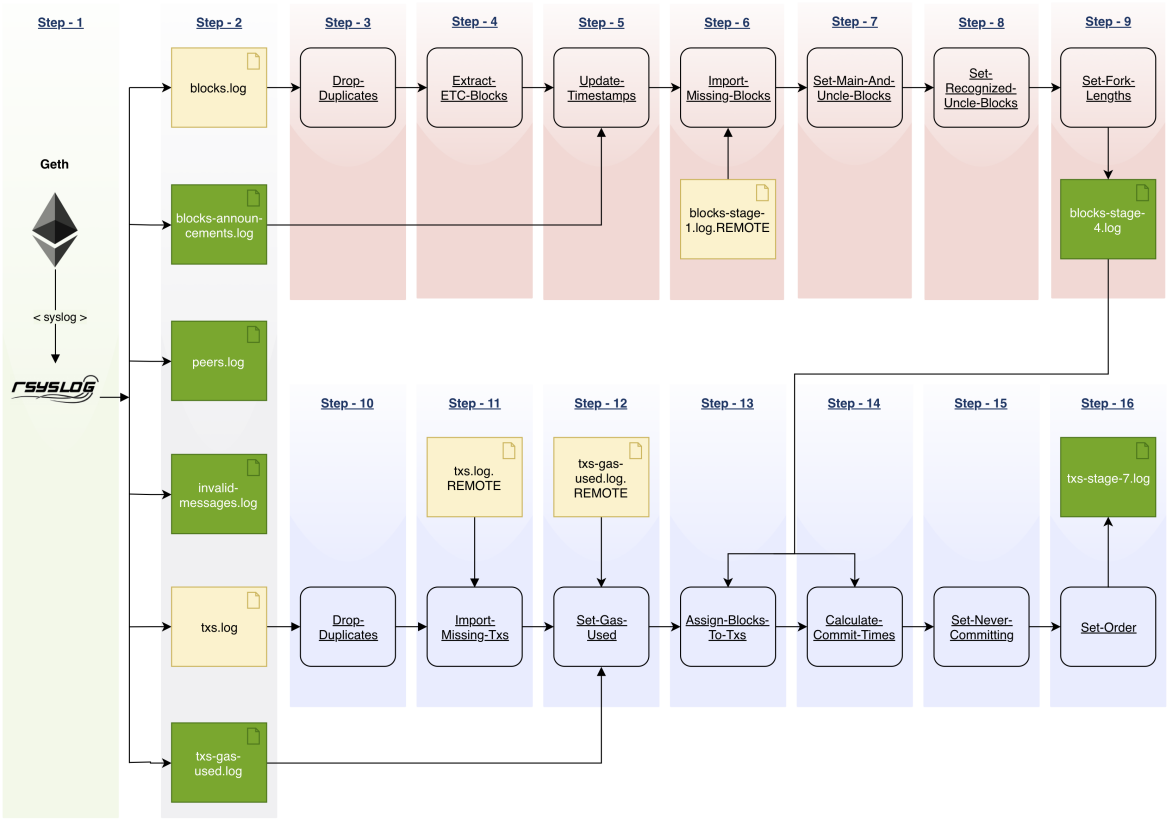


Figure 4.1: Flow diagram of log processing

4.4.1 Blocks Log

The *blocks.log* contains the receptions of messages announcing new blocks from other peers on the network.

This log, however, needs to be specifically processed and filtered before we could use it for calculating metrics. For instance, from the log in its original form one can not tell whether a particular block represents a main block or if it is an uncle. Moreover, there are many redundant receptions of the same blocks. So the first action (*Step - 3* on the diagram) is to drop all redundant block receptions. Then, in *Step - 4* we extract Ethereum Classic blocks to a separate log file and leave only Ethereum mainnet blocks. The script from *Step - 5* compares both files *block-announcements.log* and *blocks.log* and keeps only the earlier block reception timestamp for each block. It can happen that the Ethereum client or the whole server on which it runs may experience a network crash or an another failure and so may not receive new blocks during the service outage. The script *Import-Missing-Blocks* imports all missing blocks from an another Geth peer from our infrastructure and marks such blocks as "imported". Marking them is important because in some metrics it is needed to distinguish them from blocks that were not imported but captured locally, e.g. in the metrics comparing the times of reception. The following script from *Step - 7* marks blocks that make part of the main chain to distinguish them from uncles. The following script in *Step - 8* marks the uncle blocks which were referenced in some of the blocks on main chain as recognized uncles. Finally, the script *Set-Fork-Lenghts* calculates the length of each uncle block. This set of scripts produces one final *blocks-stage-4.log* log-file for each machine on our infrastructure.

4.4.2 Block Announcements Log

Blocks are propagated either directly as a whole with their bodies (these messages are captured in *blocks.log*) or only their hashes are announced in the network. The *blocks-announcements.log* captures these light announcements along with the timestamp of the message arrival.

4.4.3 Neighboring Peers Log

The *peers.log* captures timestamped connections and disconnections of peers along with the type of peer's client or the cause of disconnection.

4.4.4 Invalid Messages Log

Every single message of any type captured by Geth must pass through various validity checks.

Those messages that fail any of the checks we save along with the cause of error and with a timestamp – in *invalid-messages.log*.

4.4.5 Transactions Log

The *txs.log* contains all the new transaction receptions along with the values that describe them, such as transaction's hash, nonce, gas price, hashes of sender and recipient, etc. Some values, however, still need to be set; such as gas used, a list of blocks in which the transaction is included, information whether the transaction was received in order, and more.

Similarly as in the case of blocks, the first action to be performed is to get rid of duplicate receptions of the same transactions. This is done in *Step-10* by the *Drop-Duplicates* script. Next, in *Step-11*, we import missing transactions from another nodes from our infrastructure and mark those transactions as imported. In *Step-12* we set the gas used property for every transaction that was executed on any of our peers. The following *Step-13* assigns for each transaction the hashes of blocks in which is included. Next, in *Step-14* we calculate the times it requires transactions to become committed. In *Step-15* we mark the transactions that certainly will never commit following the algorithm mentioned in 4.2.7. In the last *Step-16* we define whether a transaction was received in order or not.

The final product of this sequence of scripts is then stored in *txs-stage-7.log*.

4.4.6 Transactions' Gas Used Log

The *gas-used* property is not one of the values that come along with each propagated transaction in the Ethereum network. Only after a transaction's proper execution the Geth client yields the actual amount of gas consumed. We log these results in the *txs-gas-used.log* log-file.

Chapter 5

Implementation and Deployment

In this chapter, we introduce the measuring infrastructure and the implementation of our Ethereum mainnet monitoring nodes.

5.1 Infrastructure

Our measuring infrastructure consists of four computing instances dispersed all over the globe. Two of them are rented virtual machines from Google Cloud; the first one (US-S1) is positioned in South Carolina - USA and the second one (CN-S2) in Taiwan. The third instance (CZ-S3) is a personal virtual machine placed in Czechia. The fourth and last machine (PT-S4) is a dedicated multicore computer located in Portugal.

These instances use NTP over the Internet for clock synchronization providing in 99 % of cases offsets lesser than 100 ms and in 90 % of cases lesser than 10 ms [MTM06].

The parameters of the machines are presented in Table 5.1.

server-ID	**CPU-type**	**#cores**	**RAM**	**SSD**	**OS**	**Network**
US-S1	Intel Xeon 2.3 GHz	4	15 GB	370 GB	Debian 9	8 Gb/s
CN-S2	Intel Xeon 2.3 GHz	4	15 GB	370 GB	Debian 9	8 Gb/s
CZ-S3	Intel Xeon 2.4 GHz	4	8 GB	500 GB	CentOS 7	10 Gb/s
PT-S4	Intel Xeon 2.2 GHz	40	128 GB	1,000 GB	Ubuntu 18.04	10 Gb/s

Table 5.1: Hardware specification of instances.

5.2 Monitoring Nodes

5.2.1 Hardware Requirements

In order to synchronize and catch up with the Ethereum's mainnet each individual peer must validate and execute all the new incoming messages with changes to the platform's state. That causes considerable amount of reads and writes which the slow HDDs cannot handle. Thus, the necessary requirement for running a full Ethereum node is having a fast SSD with enough capacity for storing the blockchain and related data which demands around 200 GB at the moment of writing this document.

In our prior tests we had ran Geth on the CZ-S3 instance with only one CPU and were not able to connect to more than 25-30 peers stably. Changing the size of RAM from 1 to 8 GB had no effect on that. After adding three more CPU cores the client had been able to connect to 100 - 150 peers which we find sufficient for the needs of our work and thus rented the Google Cloud machines with similar computational power. The PT-S4 instance with its 40 cores is our most powerful machine and is able to handle a connection with 200 or more peers at any moment. For comparison, the authors of [KMM+18] performed their tests on 32 core machines with 10 Gb/s network link and were able to measure precisely the peer latencies of the whole Ethereum network.

Our work investigates network properties with millisecond precision so we depend on an excellent network connection. Thus, we made sure to have all our instances connected directly to the Internet backbone with solid network throughput.

5.2.2 Software

Even though the hardware specifications of individual instances differ; they all run the same software configuration which essentially consists of the GNU/Linux operating system, our modified Geth Ethereum client and a specially configured instance of the Rsyslog logging daemon.

5.2.3 Ethereum Client

Our Ethereum measurement peer is based on the Geth open-source Ethereum client implementation in its latest version – which is 1.8.23[1]. We made no changes to the program's logic; the only modifications represent the addition of several syslog calls that capture all received transactions, blocks and invalid network messages as well as information about connected peers. We also prepend each of these events with a local timestamp at the exact moment of the message

[1] https://github.com/ethereum/go-ethereum/releases/tag/v1.8.23

arrival, before any message validation is done. All these changes accounted to slightly less than 1 000 of lines of code written in the Go language [2].

The client runs with the default settings except for two parameters. Firstly, we allow to connect to unlimited number of peers instead of the default 25. Secondly, we set the minimum gas price of incoming transactions to zero in order not to discard underpriced transactions. This is because we want to observe all transactions on the network for the needs of the metric measuring propagation delays of transactions. These settings are identical with the measurement client configuration used in [WGP+], allowing us to compare our results with theirs.

5.2.4 Rsyslog

The Rsyslog was configured in order to receive all syslog calls from the Geth client and to pass them into separate log files for each message type. Next, the Rsyslog performed automatic chunking of logs to 500 MB pieces as well as an automatic zipping. That was necessary because of the great number of propagated messages. Finally, it was necessary to increase the default maximal size of syslog messages from default 8 kB to 100 kB, because we have observed messages containing blocks as big as 30 kB.

5.2.5 Linux Configuration

On each machine, we have configured the firewall settings not to block incoming connections on the ports that the Geth clients listen. Next, we configured the NTP clock synchronization and set the timezones to the universal Greenwich mean time (GMT).

Finally, we configured init scripts in order to start automatically both Geth and Rsyslog in case of eventual restarts.

[2] https://golang.org

Chapter 6

Results and Evaluation

In this Chapter, we present the results of our measurements and comment the current condition of the Ethereum network.

We were listening to the network traffic of the Ethereum main network using the infrastructure of highly connected peers from April 1st to May 2nd in 2019, and collected information about 216 656 blocks (including forks) with the block numbers ranging from 7 479 573 to 7 680 658. As we have compared with Etherscan[1], we have missed 4 recognized uncle blocks out of the total 15 104 uncles from that period.

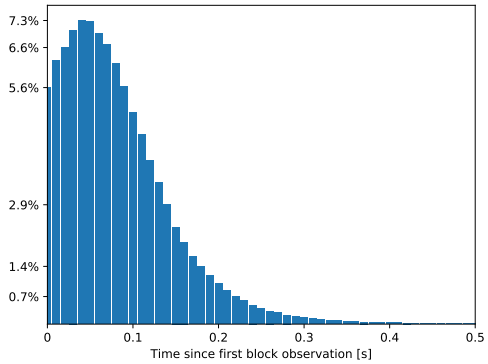
Then, we performed a complementary measurement on single peer with default settings from May 2nd to May 9th, for the needs of metric in Section 6.12.

6.1 Information Propagation

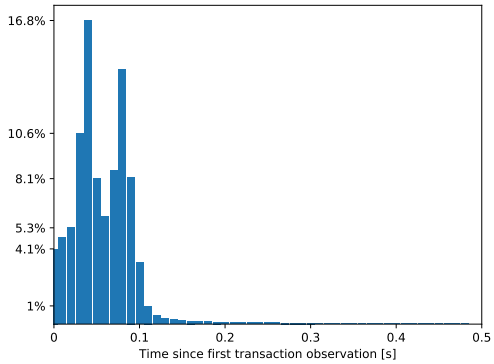
In this metric we measure the propagation delays of blocks and transactions. To achieve that, we capture their arrival times on each measurement node from our infrastructure with synchronized clocks. The delay is then the difference between the first observation of the message and the times of arrival on the remaining peers. For instance, if a transactions is observed by our four instances for the first time at times: 20, 10, 12, and 15; the the delays are: 10, N/A, 2 and 5. Finally, we accumulate all the delays and calculate their distribution.

Figure 6.1 reveals considerably low propagation delays. The median time until a node receives a transaction is just 60 mili-seconds (ms). The propagation delay of the 90 % fastest transactions is under 111 ms. Finally, it takes 0.8 seconds for 95 % and 38.5 seconds for 98 % of transactions to propagate through network. These slow transactions from the approximately 2 % tail may be problematic, e.g if by any chance there were some further transactions that depend on their

[1] https://etherscan.io/



(a) The histogram of times since the first block announcement.



(b) The histogram of times since the first transaction announcement.

Figure 6.1: Information Propagation

execution.

Blocks, in spite of their substantially greater size, propagate only slightly slower. Their median propagation delay is 74 ms and the mean is at 109 ms. In total 95 % of blocks make it below 211 ms through the network and 99 % under 317 ms.

Back in 2013, the authors of [DW13] measured the propagation delays of blocks in the Bitcoin network following exactly the same technique. Unfortunately, we do not know about any newer work replicating that measurement. Nonetheless, their results showed that in Bitcoin, the median time until a node receives a block is 6.5 seconds and the mean is 12.5 seconds. Even after 40 seconds, still 5 % of nodes do not know about the newly propagated block. The reasons for the prolonged propagation times in Bitcoin were thoroughly explained in Chapter 3. One of the most important causes is the three-round block propagation strategy that includes full verification before propagation. Ethereum, in contrast propagates light-weight block announcements directly. The second cause, back in 2013, was bad interconnection of nodes in the Bitcoin network. As the authors of [DW13] showed in an experiment, they could substantially decrease propagation times by deploying just one interconnecting node with great throughput.

We are aware that the Bitcoin's network progressed since then and so we cannot compare our results directly with [DW13]. We surely know that the second cause is no longer valid, because as the recent work [KMM+18] from November 2018 shows; the latencies between Bitcoin peers are smaller than in Ethereum.

Our measurement reveals that information in Ethereum propagates fast, and surely is not the main cause of forks, as was the conclusion of [DW13] for Bitcoin. Due to the measured results it appears that miners continue mining blocks even after discovering newer blocks. They can afford that thanks to the uncle rewards, which are almost guaranteed for forks of length one, as

we show in our measurement from Section 6.2.

6.2 Prevalence of Forks

This metric measures the total number of blockchain forks, where we distinguish between forks of lengths one, two and possibly longer ones. We also calculate the proportion of observed forked blocks and the total number of blocks.

Out of the 216 671 blocks that we captured during our one-month long measurement, 92.81 % of them became part of the main chain, 6.97 % became uncles referenced by some block from the main chain, and only 0.22 % of blocks became "so-called" unrecognized uncles. The very small number of not recognized forks leads to the greater chance for even weak miners to get rewarded.

The result is presented on Table 6.1.

Fork Length	Total	Recognized	Unrecognized
1	15 171	15 100	71
2	404	0	404
3	10	0	10

Table 6.1: Fork lengths.

We have not observed any fork longer than 3. The interesting observation is that more than 99 % of forks of length one were later recognized. All longer forks than one were not recognized.

In 2017, the same measurement was performed in [WGP+]. Since then, the proportion of uncle blocks increased by more than one percent and their lengths increased as well. We can only speculate what is the real cause of that. One explanation could be that the number of separate miners increased, which generally leads to the greater chance that two miners will propagate at the same time and create forks. In addition, back then the inter-block time, which is the mean time between two succeeding blocks,[2] was one second higher which also could had contributed.

6.3 Empty Blocks

We have observed a significant number of blocks that were empty – i.e. with no transactions. The miners of empty blocks do not waste time validating and including transactions and so can propagate them slightly faster at the cost of not getting paid for the transaction fees.

These blocks, however, are harmful for the network, because they increase commit time of transactions. This is because the transactions that could have been included in an empty block

[2] https://etherscan.io/chart/blocktime

must wait for being included in next block. If an increased number of miners switched to this strategy, it would be disastrous for the platform.

We were in interested how many blocks of this type are in the network, and secondly, whether the proportion changes if we look at main and uncle blocks separately.

The results revealed 2 921 empty main blocks (1.453 %) out of 201 086 all main blocks, and 185 empty forked blocks (1.187 %) out of 15 585 all forked blocks.

We observe, that empty blocks indeed become the part of main chain more often. The second observation is, that their number (1,43 % of all blocks) is not insignificant and is already prolonging commit times of transactions.

6.4 Transaction Commit Time

This metric measures the distribution of the time it takes for transaction inclusion which is the difference between time the transaction was first observed at a node to the time it was included in a block. Then we analyze the commit times for various numbers of subsequent confirmation blocks.

This exact measurement had been performed in [WGP+] where the authors had chosen to measure the commit times for 3, 12 and 36 confirmation. We follow them in order to be able to compare our results with theirs.

6.4.1 Results

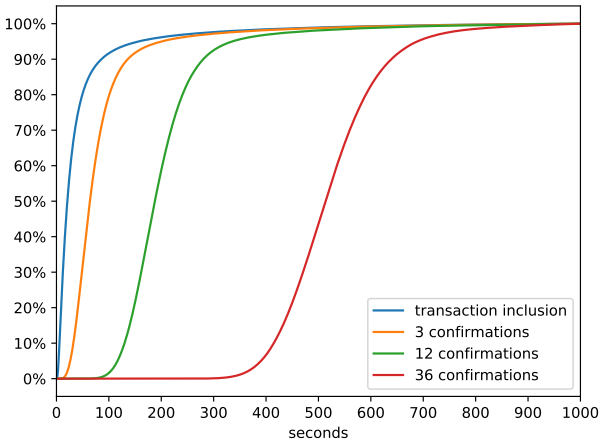


Figure 6.2: Time for transaction inclusion and commit (3, 12 and 36 confirmations).

Visually, we achieve the same results as in [WGP+], that is the curves representing commit times follow similar pattern. Moreover, on Figure 6.2 we observe that higher number of confirmations

leads to slightly less steeper curves. This indicates the increased fraction of transactions that must wait longer to be committed.

If we look in more detail at the commit times, we observe that the median waiting time for '12-block commit' is 189 seconds whilst the authors of [WGP+] measured 200 seconds. This is, however, because of the inter-block time, which decreased[3] from 14,3 s to the current 13,3 s. To prove that, it suffices to subtract 12 times the mean inter-block time from the time for '12-block commit', and in both cases we end up with 29 seconds long delay.

6.5 Transaction Reordering in Forked Blocks

We were interested whether the transactions in forked blocks and associated blocks on the main chain share the same transaction ordering. To measure that, we customized the Jaro distance metric, which we present below in Section 6.5.1.

6.5.1 Jaro metric

Jaro metric originally had been intended to measure the order of common characters between two strings. We adjust the metric to work with transaction hashes instead of characters and to perceive the whole sets of transactions included in a block as the supposed strings. The score returned by the Jaro metric is normalized such that 0 equates to no similarity and 1 is an exact match. In general, the higher the Jaro distance is, the more similar the transaction sets s_1 and s_2 are. The definition of our adapted Jaro metric is the following: for two sets of transaction hashes s_1 and s_2, let $s_1{}'$ be the hashes in s_1 which are in common with s_2, and let $s_2{}'$ be analogous. Let T_{s_1,s_2} measure the minimal number of transpositions of hashes in $s_1{}'$ relative to $s_2{}'$.

$$Jaro(s_1, s_2) = \frac{1}{3} \cdot \left(\frac{|s_1{}'|}{|s_1|} + \frac{|s_2{}'|}{|s_2|} + \frac{|s_1{}'| - T_{s_1{}',s_2{}'}}{|s_1{}'|}\right)$$

[3] https://etherscan.io/chart/blocktime

6.5.2 Results

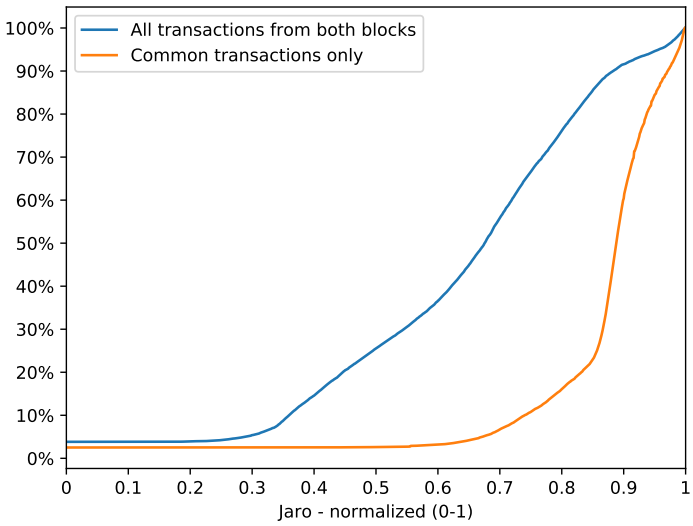


Figure 6.3: Transaction reordering in forked blocks

Firstly, we calculated the regular Jaro metric as defined in subsection 6.5.1. Secondly, we computed the same metric excluding transactions that appear only in one block. The latter measurement is meant to calculate the degree of transaction reordering only. In both cases, we performed the measurement on the same set of forked block; the 15 585 forked blocks from Section 6.2.

The median Jaro of the second measurement yields 0.8986. This shows that most of the transactions in forked blocks are in almost the same order as the transactions in associated main-chain blocks. On the other hand, the proportion of forked blocks that have transactions in exactly the same order is relatively small - 16 % only.

The line representing the first measurement is steeper which indicates that the forked blocks include significant number of other transactions. Nonetheless, the median Jaro is in this case 0.6866, which still shows decent similarity.

Finally, we calculated that approximately 3/4 of transactions from forked blocks are also in associated main blocks. More precisely, we measured that the median proportion of common transaction to the total number of transactions in main block is 72,5 % and 76,6 % in the case of forks.

6.6 Transaction Reordering

In this measurement we compare commit times of "in-order" and "out-of-order" transactions. The sender stamps every transaction with a nonce, which is incremented by 1 for each new

transaction. We say that two transactions assembled by the same sender were received out of order, when we first capture the transaction with higher nonce.

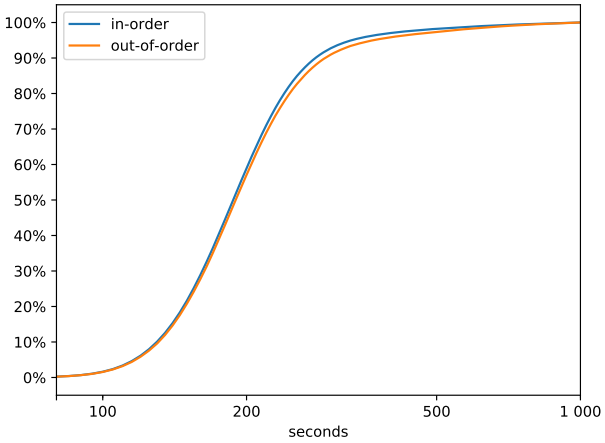


Figure 6.4: Commit delay (sec) for transaction based on ordering.

This is a replicated measurement from [WGP+] that was executed in April 2017. The authors performed the measurement on a one-day sample of logs and observed that 6.18 % out of all committed transactions during that period were received with out-of-order nonces.

We observe 11.54 % out-of-order committed transactions out of all committed transactions during our one-month long measurement. We observe, however, that the proportion of out-of-order transactions fluctuates from day to day with 4 % deviation from our one-month average. As well as the authors of [WGP+], we do not know the exact cause. We hypothesize, that there are more independent factors, such as the varying motives of Ethereum users, or the current overall network connectivity.

Our measurement reveals the prolonged commit time of out-of-order transaction. It takes 192 seconds for 50 % and 325 seconds for 90 % of out-of-order transactions to commit. For comparison, the mean time for in-order received transactions is 189 seconds and 90 % of these transactions need 292 seconds for committing. Complete results are plotted on Figure 6.4.

The authors of [WGP+] observed even more significant difference between in-order and out-of-order commit delays. The results indicate that out-of-order reception negatively affects commit times. It is not surprising because such transactions must wait for their delayed predecessors before committing.

6.7 Gas Used per Transaction Type

This measurement investigates the amount of gas consumed per transaction type. In April 2017, the same measurement was performed in [WGP+]. For this reason, we are able to comment about how the average gas used per transaction changed since then.

We distinguish between three types of transactions:

- regular value transfers
- contract function calls
- contract creations

The differentiation is done as follows: We consider a message to serve as a contract creation if the address of the recipient is empty; as defined in the Ethereum specification [Woo14]. There is, however, no consensus about how to distinguish between regular value transfers and contract function calls. In [WGP+], the authors simply say that the messages that are not contract creations, and consume 21 GWei of gas (which is the base cost of transaction), are regular transfers. The remaining messages are function calls.

We consider messages that are not contract creations and which have data field empty as regular transfers, and those transaction with some data (with a message for a contract) as message calls.

6.7.1 Results

Out of 21 960 051 captured transactions, we calculated this metric from the subset consisting of 20 654 578 valid transactions included in main blocks. There were 8 917 902 regular value transfers, 11 642 182 message calls and 94 494 contract creations.

Transaction Type	**# Transactions**	**Median**	**75th percentile**	**90th percentile**
value transfer	8 917 902	25.2	50	100
function call	11 642 182	100	250	1 000
contract creation	94 494	1 065	1 600	4 000

Table 6.2: Gas Used (in GWei) per Transaction Type.

As can be seen in detail on Table 6.2 or on Figure 6.5, the contract creations use the most gas; this is because their code consists of computationally expensive routines. The median gas used of this type of messages increased since 2017 two-fold to 1 065 Gwei, implicating that developers deploy more complex and more computationally expensive contracts. Five percent of contract creation messages consume more than 4 500 GWei, which is more than the gas limit

per block back in 2017. The current block gas limit is at 8 000 GWei while in 2017 it was only at 4 000 GWei.

The median gas used per function call is 100 GWei and 25.2 Gwei in the case of regular transfer. These are relatively small numbers leading to reasonable costs of transaction fees.

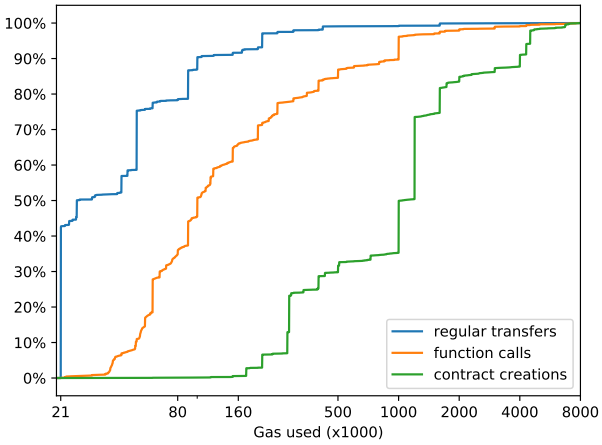


Figure 6.5: Distribution of gas usage for different types of transactions.

6.8 Impact of Gas Price

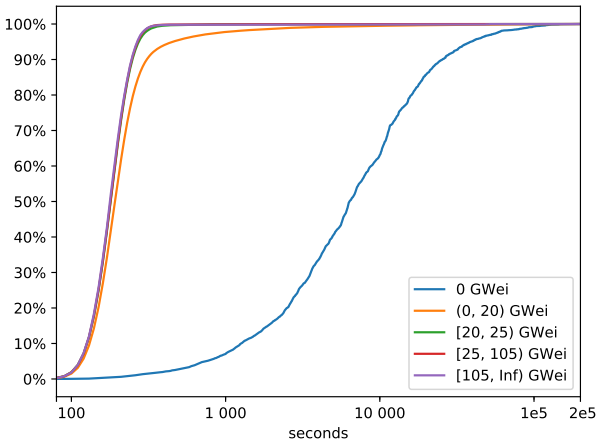


Figure 6.6: Commit delay (sec) for transaction based on gas price.

This is a replicated metric from [WGP+] measured in 2017.

We divide all observed transactions into five disjoint sets according to their gas price as follows: [0, 0], (0, 20), [20, 25), [25, 105), [105, $+\infty$) Gwei. We achieve exactly the same pattern as the original authors two years earlier, but we cannot compare the exact mean commit times because their measurement lasted 2 months longer.

We observe that transactions with zero fee commit very late (in matter of hours) as they are unattractive for miners. The transactions worth up to 20 Gwei commit significantly later than any transaction beyond this threshold. It shows, that it does not pay out setting gas price higher than 25 Gwei.

6.9 Never-Committing Transactions

Never committing transactions are those transactions which will certainly never commit. That is because there already exists another committed transaction with the same nonce which originated from the same account. These transactions are useless and only flood the network, so we were interested how many of them are there.

6.9.1 Results

Out of 21 241 139 unique observed valid transactions we find that 20 654 578 (97.2386 %) of them are already committed, 63 446 (0.2987 %) are not committed but still may commit, and 523 115 (2.4627 %) will never commit.

The transactions labeled as "may commit" are those transactions that had not committed yet, and in addition do not satisfy the condition for being "never-committing".

6.10 Duplicate Transactions

All main blocks from our one-month long measurement together included 2 0745 667 transactions out of which 1 582 423 (7.21 %) were duplicate – i.e were included in some uncle blocks as well. This metrics interests us as the Ethereum platform would benefit if this number was as low as possible. That is because duplicate transactions are useless since transactions from uncle blocks are not executed. In contrast, uncle blocks are useful because they strengthen the platform's security.

The measured number of duplicates correlates with the number of forks (7,19 %) and reveals that forks include similar number of transactions as main blocks.

6.11 Invalid Network Messages

Every message – such as block announcement, connection attempt or new transaction – received by the Geth client must pass through a validation phase before they are processed. In our

measurement, we captured all these failed verifications with the aim to observe attempts of possible flooding attacks.

At the end of our measurement, we observed on each of our machines a few messages signaling a fail due to decoding error. In addition to this, each machine received between 10 388 to 24 798 of block header messages that failed due to the DAO check – confirming so the presence of Ethereum Classic peers, that was revealed in [KMM+18].

6.12 Message Reception Redundancy

In Section 2.5, we discuss that Geth nodes propagate newly received block announcements to all their neighbors – which is by default 25 – except for the peer that sent them that message. The second most popular client Parity propagates such messages to five peers only.

We were interested what is the actual number of redundant messages that the most popular client, which is Geth, with defaults settings receives. Thus, we deployed a node, that was connected to the default number of peers (25), and listened to the network between May 2nd and May 9th in 2019.

For comparison, we also provide this metric based on the data from our one-month lasting measurement captured by our highly connected Geth client running on the Portuguese server, which was always connected at least to 150 peers, and on average to 200 peers.

6.12.1 Results for Geth with 25 peers

Blocks are propagated through two types of messages: Either in the form of light announcements consisting of the block's hash only, or they propagate directly, including both header and body.

Block Announcements

The average number of redundant receptions of all block announcements was 2.585 with median at 2 receptions. 10 % of the most propagated announcements were received 5 times, and 1 % was received 7 times.

Whole Blocks

The average number of redundant receptions of all full blocks was 7.043 with median at 7 receptions. 10 % of the most propagated new-block messages were received 10 times, and 1 % most redundant receptions was at 12.

Both messages combined

If we count both types of block propagating messages together, we achieve the following numbers: The average number of redundant receptions was 9.11 with median at 9 receptions. 10 % of the most propagated blocks were received at least 12 times, and 1 % was received 15 times.

6.12.2 Results for Geth with 200 peers

Block Announcements

The average number of redundant receptions of all block announcements was 8.64 with median at 6 receptions. 10 % of the most propagated announcements were received 17 times, and 1 % was received 59 times.

Whole Blocks

The average number of redundant receptions of all full blocks was 66.88 with median at 68 receptions. 10 % of the most propagated new-block messages were received 101 times, and 1 % most redundant receptions was at 123. The single most frequently received whole block was captured 159 times.

Both messages combined

When counting both types of block propagating messages together, we achieve the following numbers: The average number of redundant receptions was 73.59 with median at 75 receptions. 10 % of the most propagated blocks were received at least 109 times, and 1 % was received 139 times.

6.12.3 Discussion

How to decide, whether the measured numbers of redundant receptions are too low, too high or optimal?

The authors of the highly cited paper [EGKM04] suggest that in networks with failures, it is enough for the epidemic broadcasting strategies to disseminate information to a logarithmic number of neighbors with respect to the total number of peers.

According to the latest estimation from [KMM+18], there are around 15 000 peers, so the propagation strategies should propagate to at least ln(15 000) of peers - which equates to ~ 10 peers.

Our results for Geth with default number of peers reveal the median and mean number of redundant block message receptions to be 9 in the former, and 9.11 in the latter case.

Chapter 7

Conclusion

This chapter begins with a list of this work's most notable achievements. Then we discuss the current state of the Ethereum platfom from the network perspective. Next, we propose our suggestions about how to improve Ethereum client implementations. Finally, we present two suggestions about complementary measurements to this work.

7.1 Achievements

- We have measured a wide range of Ethereum's network properties that may have the potential of directing the aim of future improvements of the network protocol of Ethereum as well as similar platforms.
- By replicating measurements performed in 2017, we were able to get insights on how the network's behavior changed since then. The major differences since then are: increased number of forks, slightly increased transaction commit times and twofold increase of the average computational expensiveness of deployed smart contracts.
- We designed the transaction reordering metric and showed that almost 90 % similarity of transactions' ordering included in main blocks and associated forks.
- Our measurement nodes captured the presence of peers from the Ethereum Classic network which are not only useless, but cause significant network overhead. In Section 7.3 we propose a possible solution for this problem.

7.2 Ethereum's Performance

Our information propagation measurement captured the excellent propagation times of blocks and transactions. We measured the average delay between the times of the newly mined blocks

receptions on our infrastructure consisting of four geographically distant nodes to be on average at 109 ms. This indicates the very good connectivity of the platform.

Our other measurements showed mostly positive results as well. We, however, must be cautious about premature conclusions about the state of the network

The Ethereum platform does not scale with increased number of participants as we discussed in Chapter 3. The problem is rooted in the platform's basic functioning, where every participant must process all information of the whole platform.

In the end of 2017, when the demand for Ether increased, the capacity of blocks was not sufficient to include all transactions which lead to extremely prolonged commit times, skyrocketing transaction fees, and eventually to the lost faith in the technology by many of investors[1].

In the last two years, the block utilization in most of times ranged[2] between 70 % to 90 %, thus even a slight increase of the amount of transactions in the network may lead to the same consequences as the events from the end of 2017.

This is because Ethereum is not just a simple cryptocurrency, but a complex decentralized platform for smart contracts and so the amount of information transferred and processed is substantially higher than, for instance, in Bitcoin when assuming the same number of peers.

7.3 Ethereum Improvement Directions

Ethereum, in order to remain usable even with increased number of users, must address the scalability issue in the long term. The developers are intensively implementing "sharding"[3], as we discuss in Section 3.2.1, but that will take long time because of its enormous complexity.

As a reaction to the observation of the Ethereum-classic traffic in the non-classic network, we propose an easily implementable improvement to the Ethereum client implementations that would substantially decrease the amount of network traffic and increase performance of nodes.

Both platforms run on the DEVp2p underlying network and share the same network ID as well as the genesis hash. This is because they used to act as one network, before they forked back in July 2015. The problem of current client implementations, as is explained in [KMM+18], is that they maintain connection with Ethereum-classic peers and thus receive from them useless data.

We observe, that upon each new connection, the peers exchange a set of messages, including the information about last block headers out of which it is possible to identify Ethereum-classic peers. We suggest to exploit that and disconnect from those peers.

[1] coinmarketcap.com/currencies/ethereum/#charts
[2] etherchain.org/charts/averageBlockUtilization
[3] github.com/ethereum/wiki/wiki/Sharding-roadmap

We propose to add the three following methods to the logic of all Ethereum client implementations:

1. Immediate disconnect after observing the "DAO fork" in the "BlockHeaders" message exchanged after a connection.

2. Maintain a local persistent table of "enode" identifiers of Ethereum-classic peers and add there the peers from Step 1.

3. Before connecting to new peers, first verify whether they do not belong to the table of Ethereum-classic peers.

The only weakness that we see is that this strategy would not disconnect from not yet synchronized Ethereum-classic peers whose latest block is before the DAO fork.

7.4 Future Work

Regarding possible future work, we present two suggestions:

- Our measurements were performed on the Geth client implementation which is only one out of many types. We advocate to replicate the measurements capturing network properties, such as those measuring redundant message receptions on other client types to observe if their behavior differs. In addition to this, it would be worthwhile to make a comparative study of the most prominent client implementations. The Ethereum users would appreciate knowing which implementation is the most efficient as well as the developers of the other implementations could get inspired what to improve.

- We suggest to perform our reception redundancy measurement with the aim at transactions instead of blocks. Both types of messages follow slightly different propagation strategies and their properties, such as their average size, are very different.

Bibliography

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust (pp. 164-186)*. Springer, Berlin, Heidelberg, April 2017.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer Berlin Heidelberg, February 2016.

[CV17] Christian Cachin and Marko Vukolic. Blockchains consensus protocols in the wild. In *arXiv:1707.01873*, July 2017.

[DGHK17] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *arXiv preprint arXiv:1702.04467*, 2017.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference*, pages 1–10, September 2013.

[EGKM04] Patrick Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. From epidemics to distributed computing. In *IEEE Computer*. IEEE, 2004.

[ES14] Ittay Eyal and Emin Gun Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security (pp. 436-454)*. Springer, Berlin, Heidelberg, March 2014.

[FJJ+01] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. A global internet host distance estimation service. In *Networking (TON), pages 525–540*. IEEE/ACM, 2001.

[GBE+18] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Rober Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. In *Financial Cryptography and Data Security 2018*, January 2018.

[KMM+18] Seoung Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. Measuring ethereum network peers. In *IMC '18 Proceedings of the Internet Measurement Conference 2018*, pages 91–104. ACM, November 2018.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarte. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (pp. 254-269)*. ACM, October 2016.

[MLP+15] Andrew Miller, James Litton, Andrew Pachulski, Neal Gupta, Dave Levin, Neil Spring, and Bobby Bhattacharjee. Discovering bitcoin's public topology and influential nodes. May 2015.

[MTM06] Cristina Duarte Murta, Pedro Rodrigues Torres, and Prasant Mohapatra. Characterizing quality of time and topology in a time synchronization network. In *GLOBECOM*, 2006.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[NF19] Jianyu Niu and Chen Feng. Selfish mining in ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Januar 2019.

[NG16] Christopher Natoli and Vincent Gramoli. The blockchain anomaly. In *Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium*, pages 310–317. IEEE, October 2016.

[Pop16] Serguei Popov. The tangle. 2016.

[SH17] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *arXiv preprint arXiv:1702.05511*, 2017.

[SP17] Gideon Samid and Serguei Popov. Bitflip: A randomness-rich cipher. April 2017.

[SZ15] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *Financial Cryptography and Data Security - 19th International Conference*. FC, 2015.

[Vuk15] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.

[WGP+] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On availability for blockchain-based systems. In *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium*, pages 64–73. IEEE.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.