Analyzing the Peer-to-Peer Network of Ethereum

By

Egbe Michael Eyong

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of

Computer Science and Networking offered jointly by the University of Pisa and

Sant'Anna School of Advanced Studies

Supervisor: Associate Professor Laura Ricci

October 4, 2019

Abstract

Ethereum is a blockchain-based computing platform for building decentralized applications. For Ethereum to succeed in its core ideology of simplicity, universality, agility, modularity, non-discrimination, and non-censorship [27], it needs to employ several secure decentralized data systems. These decentralized systems are based on a peer-to-peer (P2P) computer network and are not subject to the whims of a central authority like government or server administrators. Valued at over 22 billion US dollars as of September 2019, it is the second largest Blockchain-based cryptocurrency by market capitalization after the Bitcoin [12]. While the P2P network of Bitcoin has been given lots of research efforts [54, 55, 56, 57, 58], Ethereum commonly referred to as Bitcoin 2.0 with a fundamentally diverse and complex network structure has received less comprehensive study. The purpose of this thesis is to understand the P2P network of Ethereum. We did an overall introduction of the Ethereum blockchain system to provide background study. We then analyzed the P2P structure of the Ethereum Network over a period of two months on the Mainnet - Homestead. Data were collected from over 217,514 distinct nodes. Collected data allowed us to make analysis on the network types, node status, service type of nodes, node geographical distribution and ISP providers.

Acknowledgements

I would like to express gratitude to my thesis supervisor Associate Professor Laura Ricci for her insightful scholarly input and feedbacks, patience, guidance and motivation in developing this thesis. Further, I would like to thank my family for providing me with endless support and encouragement throughout my years in school and through the process of researching in writing this thesis. This would not have been possible without them.

Table of Contents

CHAPTER 1

Introduction

The need for decentralized cash system has been exploited more as a theoretical concept until Satoshi Nakamoto's famous Bitcoin white paper [1]. The paper made ascertains, the first concept of a purely peer-to-peer version of electronic cash systems, managed autonomously in a decentralized way. The blockchain represents the underlying technology that enables this system. It is a public ledger where digital files such as lists of transactions or contractual agreements are combined into ordered blocks and stored on a distributed database. Each block is timestamped and cryptographically linked to the previous one. This complex cryptographic layer coupled with timestamping servers and a P2P network make records of transaction immutable. The tamper-resistance or immutability, centralization, anonymity, and security are the core reasoning behind the blockchain. Therefore, its utilization has spread across many applications and use cases. While the best-known use case of blockchain is as a digital asset (cryptocurrency), it has also been proposed to self as the basis for medical records [72], domain registration [73], and assets registries [74]. Its immutability, consensus, provenance and finality attributes make it more suitable for supply chain businesses [75], e-voting [76], auditing and compliances [77]. Other use cases extend to decentralized applications making use of smart contracts. Smart contracts represent programs that can automate extremely complex applications without the possibility of downtime, censorship or third-party interference [5, 27]. These contracts have been paired and deployed with IoT for

device-to-device payment [60, 79], machine maintenance [80], privacy management [63], notary [60], shipments tracking, energy applications, games, just to name a few. Blockchain intriguing possibility of recording transactions, establishing contracts – in the case of Ethereum, and maintaining identity in a decentralized way gave rise to a market capitalization of 286 billion US dollars as of September 2019 [12]. Bitcoin and Ethereum are respectively the first and second most valuable application of blockchain technology.

Though Bitcoin and Ethereum share the same fundamental principle of a distributed ledger and cryptography, they differ in many ways. For instance, Bitcoin is a stack-based language while Ethereum is Turing-complete. Any programming language able to compute anything computable given enough resources is said to be Turing-complete. The Turing-completeness of Ethereum has led to the applications of smart contracts. These contracts provide an abstract layer to anyone to formulate their own rules for transaction ownership, employing a set of cryptographic rules that are executed when certain conditions are met [6]. Other differences between Bitcoin and Ethereum relate to the block time; Ethereum transaction is approved in seconds whereas Bitcoin transaction is in minutes. Also, while Bitcoin limits the block size to 1MB, in Ethereum the block size is capped by the gas limits [9, 27].

In this thesis, we analyze the decentralized P2P network of Ethereum. We noticed several pieces of research highlighting Ethereum Smart contract applications [48, 59, 60, 61, 62, 63, 79, 80]. There is on top of that plenty of work on the consensus mechanism on Ethereum ranging from the byzantine fault tolerance [31] to the proof-of-work [29, 32], and most recently proof-of-stake [70]. Moreover, they are surveys on the security, the vulnerability of nodes [28, 37, 48], the disruptive effect forks can cause on the stability of such systems [78], sybil and eclipse attacks which target the P2P network [14, 28, 49]. Despite all this work, none relate to the information propagation and the types of messages exchanged in the Ethereum P2P network. To the best of our knowledge, this work represents one of the first efforts to analyze the detail P2P structure in terms of message types, messages exchanged, and block propagation in the network. This work is significant as the success of the Ethereum blockchain is owed to the innovative use of a P2P network that allows the creation and transfer of assets between account without relying on any intermediary.

Understanding of the P2P network structure is critical for better network design, consensus implementations and security improvement that could help in the better deployment of smart contracts.

The Ethereum P2P Network is a tale of two stacks; UDP for discovery and TCP/IP for P2P communications. To make the P2P protocol complete, a discovery process allows peers to instantly discover more nodes that support and run Ethereum client. Ethereum clients are software applications used to gain access to the Ethereum network. The discovery process is used to construct the P2P network. The node discovery process for the Ethereum Network stack is based on a slightly modified version of the Kademlia routing protocol. Kademlia is a distributed hash table (DHT) carefully built for P2P filesharing systems [25]. Ethereum discovery protocol omits the store and find_value remote procedure calls, and simply make use of the DHT to accumulate information about peers [14]. Nodes in the P2P network are identified by a 256-bit cryptographic hash public key, which is used for the Kademlia distance metric. More details of Kademlia and discovery process in chapter 3.2. The TCP stack encompasses RLPx for node discovery and routing, DEVp2p for application session establishment, and finally, Ethereum subprotocol for Ethereum specific operations. RLPx includes a two-phase handshake process on first connection to augment Kademlia lack of support for a secure connection. P2P communication between nodes running Ethereum is designed to be governed by the DEVp2p wire protocol. The subprotocols which runs on top of DEVp2p are self-contained and define sets of messages transported on the network. These subprotocols are used to retrieve and store information on the blockchain. We made use of the RLPx and DEVp2p protocols to find and carefully analyze node information. In marked contrast to other P2P blockchain networks like Bitcoin and Ripple [54, 84], the Ethereum network supports multiple coexisting protocols and blockchains. This flexibility in design means there is sometimes conflict between the main blockchain networks and other test networks running Ethereum services, i.e., the protocol does not provide isolation between different Ethereum networks. We have found the DEVp2p supporting multiple services which could potentially disrupt the normal operation of the deployed local node running the Main network (network Id 1). Evidence of this from our study is the 'useless peers' (peers that deliver un-

reliable data) messages which constitute one of the main reason for a disconnection in the Mainnet network. Other instances include the reception of frequent *hello* and *status* messages from test networks – e.g., Ropsten, Rinkeby and Kovan, and other major networks – e.g., Tomochain and Egem, prompting the local client to send incompatible protocol messages. 'Hello' and 'Status' are handshakes messages normally initiated to establish a secured peer connection.

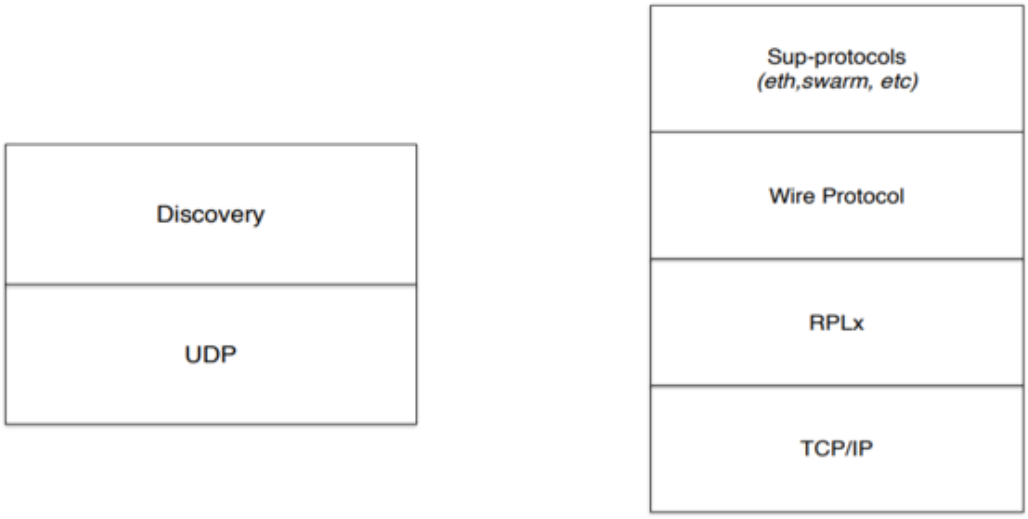

Fig1. The Ethereum P2P Network consist of two stacks; UDP for discovery and TCP for peer-to-peer communications. (source: https://slack-files.com/T9C7VSRBN-FAGT1F2BX-3844bac61e)

To run and collect data on Ethereum, we made use of the EthereumJ library/client – it is the official Java implementation of the Ethereum Network protocol. We first deployed an EthereumJ client with default configuration properties on the Mainnet Network from March 18^{th} - March 22^{nd}, 2019 to get a grasp on how the network behaves. Based on the initial results, we subsequently modified and pushed the protocol to its limit with unilateral changes to the client behaviour. A second deployment is then made for a period of 60 days (March 23^{rd} – May 23^{rd}, 2019). Collected data provide information on the size of the Ethereum P2P network, client types, network types, disconnection reasons, node geographical locations and ISP providers.

The contributions of this thesis are as follows:

1. Understanding of the Ethereum Network. To this regard, we provide an

extensive discussion on the three protocol that constitutes the Ethereum P2P network; RLPx; DEVp2p, and finally, Ethereum subprotocols. We also thought it useful to detailed background study of the consensus mechanisms and how Ethereum solves the scalability trilemma of decentralization, security, and scalability;

2. An analysis of the information propagation, the peer traffic volume, and reasons for any peer disconnections. We equally made analysis on the block generation rates;
3. Classification of the clients, network types and subprotocol services;
4. The geographical distribution of Ethereum nodes on the Mainnet network and the major ISP providers.

The rest of this thesis is arranged as follows. Chapter 2 reviews some prior research efforts. Chapter 3 goes further to describe the three protocols that constitute the P2P structure of Ethereum Network and some additional information on the consensus and sharding of Ethereum blockchain. In chapter 4, an overview of the client software used to gain access to the Ethereum ecosystem is discussed. Details of the implementation and data collection procedure is discussed in chapter 5. In chapter 6 we present and discuss our results. Chapter 7 concludes the paper and provides some hints for future work.

CHAPTER 2

Related Work

The idea of cryptocurrency is not an unfamiliar one [50] but not until recently it has been effectively implemented with the used of the blockchain technology. The concept of blockchain was initially introduced by Bitcoin to combat the double spending problem associated with digital assets in a permissionless P2P settings [1]. Ethereum key value is to provide decentralized consensus on authorized transactions and smart contracts [9, 27].

Kim et al. [3] develop a monitoring tool based on Geth version 1.7.3, to measure the Ethereum Network at scale and illuminates the properties of its nodes. Geth is the Golang official implementation of Ethereum protocol. Firstly, their tool ignores the peer connection limit at both the DEVp2p and Ethereum subprotocol layers in order to continuously discover and connect to new peers. Ethereum clients normally accept incoming connections until the max peer limit is reached (various clients have a different value for this limit). Secondly, their tool disconnects from peers as soon as it is done collecting its information in order to free slots for more peer connection. Lastly, nodes gotten from the discovery process are placed in a "static dial" list where they were periodically redialed every 30 minutes, and stale addresses of more than 24 hours ejected. Their tool uncovered a cluttered network that contains thousands of nodes running various non-Ethereum services. They affirmed a noisy DEVp2p

protocol were fewer than half of all nodes contributes to the Ethereum main network. More isolated comparisons with Gnutella, Bitcoin and BitTorrent show Ethereum differs both in size and geographical location. To this end, they calculate latencies based on smoothed round-trip times measured on direct connections. Ethereum demonstrated lower latency than Gnutella, but a higher average latency and broader distribution than Bitcoin. Lastly, they uncovered 50% of Parity (another Ethereum client) and 75% of Geth clients to be below 3 months which is like BitTorrent network in early 2007. While the authors of [3] used Parity and Geth clients to measure the Ethereum network, we have deployed the EthereumJ client to gain in-depth knowledge of the Mainnet network. Further, in contrast to the work done by these authors, we did not deliberately terminate a peer after we had collected its information. Rather, we maintain the connection to have a broader view of the node behaviour. We additionally did not allow unlimited peer bond; we limited the number of connected peers to 50 for the first deployment and later enhanced the value to 100 with a more isolated increase in the max open files to 1024 to allow room for more connections without constraint on the local hardware.

In [78], the authors made an analysis on the impact of forks on the stability of the P2P network regarding the security and market value of Ethereum (ETH) and Ethereum Classic (ETC). Their findings showed forks cause drastic partition leading to a sudden loss of approximately 90% of nodes in ETC. A network partition here means nodes can no longer communicate due to a portion of the nodes adapting to a new protocol. Significantly, they state that stabilization following a fork can take days to occur after which forks could persist with divergent behaviour. Another critical observation they noted was the unintentional security vulnerability introduced after a fork as malicious nodes could rebroadcast transactions into the other network. Regarding the marketization of both networks, they observed an efficient operation in the markets of ETH and ETC after the fork with mining rewards closely identical.

Gencer et al [4] detailed a comparative assessment of decentralization in Bitcoin and Ethereum. They relied on a novel measurement technique using information from the Falcon Relay Network and the application of well-established internet measurement techniques. Their results showed Bitcoin has a higher capacity network than

Ethereum but with more clustered nodes likely residing in datacenters. Their paper assert Bitcoin can increase the bandwidth requirements of nodes by a metric of 1.7 and nevertheless maintained the same level of decentralization as 2016. They also assert Ethereum has lower mining utilization than Bitcoin and would benefit from a relay network, and small miners experience more volatility in block rewards in Bitcoin than Ethereum.

Lago et al. [86] investigated the processing cost of an Ethereum blockchain in a small private network. They analyzed the behaviour of the network using three operations of varying complexities; linear, data manipulation and sorting algorithm. The impact of the gas (computational cost) consumption, as well as the input parameter of the algorithm, was performed to test the processing limit of the blockchain on these operations. They observed algorithms that used the blockchain for data processing are inefficient as some operations took so long to process a request. Further sorting algorithms on smart contracts were by far the costliest averaging 2.8 Mgas in 4 seconds of processing on Bubblesort ordering contracts. They assert linear processing is bad and the use of parallel processing of data might be advantageous. Unpredictability could profoundly influence the implementation of parallel transactions. Transactions in Ethereum scenario could represent the creation of smart contracts, some cryptocurrency transfer, or the invocation of smart contracts, etc. Nadi Sarrar [87] conducted an empirical study based on all blocks and transactions in Ethereum to develop an actual transaction execution constraint. They used the heterogenous earliest finish time [88] scheduler and compared their results with grand data from prior research of a simple scheduler that already achieved speedups of 1.77 and 1.94 respectively in 2017 and 2018. The technique utilized was able to speculatively run transactions in parallel with a mechanism to resolve conflicts as they occur. Hitherto, it assumes the scheduler possesses future knowledge of the account the transaction touches – this is not possible with the current version of Ethereum. Their results were found to be better than prior scheduler which made them conclude the opportunity of executing Ethereum transactions in parallel is safely increasing with time.

The authors of [89] provided an analysis of the complex network representing Ethereum blockchain transactions to understand how blockchain changes in time. They con-

sidered the flow of transaction happened in the blockchain as a network, with nodes represented as Ethereum contracts. Each transaction represents a new link in the network. The rationale behind their studies is that complex networks provide an appropriate way to model a blockchain as a complex system, together with a powerful quantitative measure to capture the essence of its complexity. They varied the number of blocks considered and obtain a different network of varying size and complexities. They argued it influenced the structure of the network. They noticed many of the nodes had a low number of links (lower degree) as a result of poor interactions in the blockchain, there was also a significant number of hubs with a higher degree. Their work is important to recognize the main contributors of the blockchain. Though the nodes with a higher degree are important, they exhibit a lower level of anonymity.

The Security Research Labs (SRLabs) released a report detailing the security impact of unpatched Ethereum clients on the stability of the network [22]. The report affirmed unpatched clients posed a 51% attack risk. Their report is based on data from ethernodes.org and it indicates many nodes using the most popular clients i.e., Geth and Parity have been left exposed for 'extended period' after patches for security flaws have been released. The reports site vulnerabilities in Parity clients that can open nodes up to being crashed remotely. The report states "According to our collected data, only two thirds of nodes have been patched so far. Shortly after we reported this vulnerability, Parity released a security alert, urging participants to update their nodes." While Parity suffers from high complexity and not all updates are included. The report indicates a critical patch scenario for Geth clients too. The SRLab issued a security-critical update for Geth nodes below v1.8.x. The report warns"If a hacker can crash many nodes, controlling 51% of the network becomes easier. Hence, software crashes are a serious security concern for blockchain nodes (unlike in other pieces of software where the hacker does not usually benefit from a crash)." The report strongly encourages the need for an automatic mechanism to update clients as early as possible.

Feld et al. [81] present an eclipse attack they argue result from vulnerabilities in Ethereum adoption of the Kademlia routing table which target the P2P network for neighbor discovery. This attack was successful on Geth software of Ethereum

node. It is realized after a victim node restart and it is based on flooding the node discovery table with Sybil nodes. Part of these attacks is by table poisoning and the monopolization of the TCP connections. Their attack was successful with only two hosts, each with a single IP address. A trivial countermeasure they proposed against connection monopolization is to enforce an upper limit on the number of incoming TCP connections, forcing clients to make a mix of incoming and outgoing TCP connections. All things considered, their countermeasures which claimed to have tended the bar (rather than controlling just two IPs, an attacker will be forced to control thousands of IP addresses to launch such an attack) for eclipse attacks was implemented in Geth version 1.8. Shortly after these countermeasures were implemented, the authors of [85] proposed a "false friend" attack that targets the Kademlia-inspired peer discovery of Geth clients, and still requiring just a minimum of two IP addresses from a distinct /24 subnet to be successful. In marked contrast to the previous attack, their empirical measurements showed this attack can be successfully mounted without assuming the node reboots at some point. Instead of overwriting the complete discovery table with Sybil nodes like what [81] did, they subtly insert adversarial nodes with carefully selected node IDs, exploiting the interplay between peer discovery and connection management. As countermeasures, they proposed: raising the outbound connections in Geth from 25 to 50; selecting nodes uniformly at random from the sets of all nodes in the table instead of just the bucket heads; throttling the inbound connection attempts; and finally, it was suggested to enforce a subnet restriction on the lookup-buffer for peer-selection.

Wüst et al. [49] presented three vulnerabilities affecting the Ethereum blockchain network and clients. In preference, they listed an eclipse attack which allowed the attacker to partition the P2P network structure by ambushing the block synchronization phase without monopolizing the connections of the victim. Since a node will never attempt to synchronize with more than one node, this means if an adversary connects to the node, the node will be unable to synchronize with a valid chain once it misses a block. Thus, keeping the node from receiving the valid block if the attack persists. Another attack outlined an exploit to force a node to accept a longer chain with lower total difficulty than the main one. An attacker could mine a longer chain than the valid chain by at least 256 blocks and later adver-

tises with a higher total difficulty. A newly instantiated node synching with the attacker won't be able to synchronize with other non-malicious nodes since its chain is at least 256 blocks longer. The last attack targets a bug in Ethereum's difficulty calculation which affects Ethereum Geth client version (at least up to v1.4). This attack resulted from a flaw in the total difficulty calculations in those versions and it is trivial to fix. Atzei et al. [48] did a survey of the attacks on Ethereum smart contracts. They group the vulnerabilities into three classes based on where they are introduced: Solidity, EVM, and blockchain. Their analysis highlighted a common cause of insecurity between smart contracts is the difficulty in detecting mismatches between intended behaviour and the actual one.

Consensus protocols constitute the backbone of this distributed ledger system and it is currently debated actively in literature [40, 41, 42, 43, 44, 45, 46]. Several pieces of research have highlighted the Byzantine fault tolerance as a key security issue on blockchain which utilizes the proof of work (PoW) mechanism for consensus [51]. Sankar et al. [52] did a comparative analysis of stellar consensus protocol (SCP), Corda and hyperledger consensus. SCP is a federated Byzantine fault tolerance (FBFT) with the concept of quorum slices. SCP idea of quorum provides asymptotic security and flexible trust making it more acceptable than earlier consensus protocol utilizing FBFT like ripple [53]. With Corda, the two key aspects to achieve consensus are transaction validity and transaction uniqueness. Hyperledger is a distributed ledger platform for running smart contracts started by the Linux Foundation. Since it proposes a practical Byzantine fault tolerance (PBFT), chaincode transactions must be deterministic in nature, otherwise different peers might have different states. SIEVE protocol is used to filter out the non-deterministic transactions, thus assuring a unique persistent state among peers.

Several studies have been made on the structure of Bitcoin P2P network. Donet et al. [56] did 37 days analysis of the decentralized P2P structure of Bitcoin Network. Their result emphasized on the almost homogenous spread of the Bitcoin nodes with a generally acceptable latency for miners to work on new blocks. Koshy et al. [54] made an analysis of the anonymity in Bitcoin using P2P traffic. Their technique mapped bitcoin addresses directly to IP data using real time transaction traffic. Former modelling of information flow in Bitcoin network have been studied in [55]. The

report asserts propagation delay as the primary cause of forks in Bitcoin network. The authors in [81] introduce a framework that traverses Bitcoin's P2P network and generates statistics regarding its size and distribution among autonomous systems. The paper affirmed there were over 10,500 publicly available peers that constitute the Bitcoin core network. The peers were distributed on more than 1,700 AS with only 10 AS holding as much as 30% of all routable peers. In addition, their result showed over 900 autonomous systems that contained just a sole peer. Their result also showed an average peer-list contained addresses that mostly resides in the same autonomous system.

CHAPTER 3

Ethereum Peer-to-Peer Network

This chapter provides some Ethereum basics and an overview of the three protocol which constitute the Ethereum P2P Network; RLPx for node discovery; DEVp2p for application session establishment, and finally, Ethereum subprotocol for Ethereum specific operations. We further provide details on consensus and sharding.

3.1 Ethereum Basics

Ethereum consists of two essential components. The first is a Turing-complete virtual processor called the Ethereum Virtual Machine (EVM), designed to be completely isolated from the environment and the rest of the network [2,7]. The second is a token of value called ether, which serves as the network currency. Ethereum Foundation explains that its ether, is not only intended to be a currency - it is a byproduct of a much grander vision; fuel for operating a "world computer" [5]. The ether is the processing power needed to produce contracts and its digital asset (cryptocurrency) serves as payments for the realization of contracts [8]. Ethereum pricing mechanism is based on gas – is the lifeblood of the Ethereum ecosystem. Gas is a unit that measures the amount of computational effort to execute a certain operation [9]. This pricing mechanism is used to optimize the resources of the network and to prevent distributed denial of service (DDoS) attacks. [28].

3.1.1 Blockchain

As earlier mentioned, a blockchain is a linked list of ordered blocks. Each block is linked with the previous by a data structure called hash pointer. A hash pointer stores information together with the cryptographic hash of the information. This hash function must demonstrate the intrinsic properties of collision-resistant, hiding, and puzzle-friendliness [65]. As a result, if an intruder modifies data anywhere in the blockchain, it would result in the following hash pointer being incorrect and therefore invaliding the adjacent block. This tamper-resistance is a core property of blockchain. Ethereum further builds on a Merkle tree using this hash pointer. A Merkle tree represents a binary tree with hash pointers coined by Ralph Merkle [64]. Blocks are grouped into pairs with the hash of each block stored in the parent node. The parent nodes are further grouped into pairs and their hashes stored one level up the tree. This goes further until the root node – genesis block. The Merkle tree is used in blockchain to proof membership of a block. In this manner, one only needs to indicate the blocks all the way up until the genesis block.

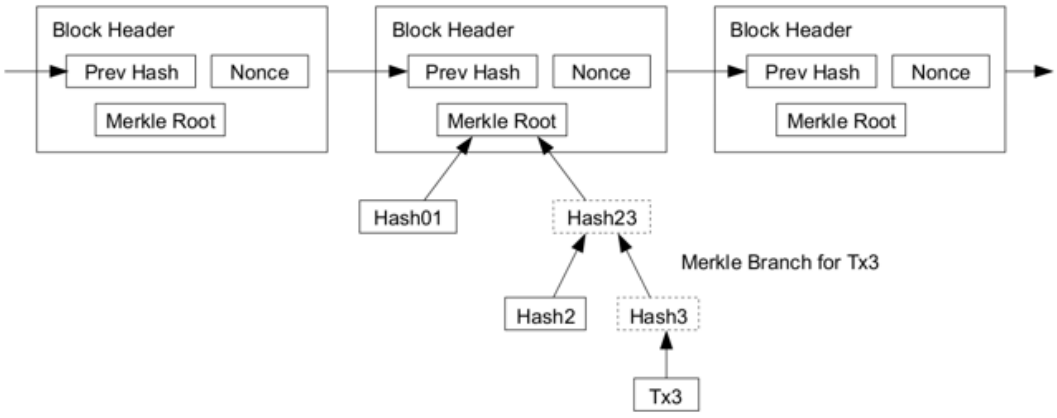

Fig2. Ethereum blockchain Architecture (source: https://ethereum.stackexchange.com/questions/268/)

3.2 RLPx Node Discovery

RLPx is a protocol suite for node discovery and communication among Ethereum nodes, based respectively on user datagram protocol (UDP) and transmission control protocol (TCP). This suite is geared towards building a robust transport, well-formed network, and software interfaces to provide an infrastructure which demand the high standards of distributed applications like Ethereum. RLPx maintain the

P2P overlay network with the means of a modified version of the Kademlia protocol. Kademlia for node discovery protocol aimed at discovering RLPx nodes to connect to. It consists of four packet types: *ping, pong, find node, and neighbours.*

3.2.1 Kademlia routing table

Kademlia is a P2P distributed hash table based on UDP [13, 25], for distributed nodes to store and retrieve data. It provides an infrastructure for millions of computers to self-organize into a network, communicate, and share resources between themselves without the whim of a central server. The distributed nature of Kademlia means there's no absolute truth where node Ids are mapped to their address (i.e., it is decentralized). Hence, each node must keep this mapping for a subset of nodes on the network in its own routing table. Kademlia minimizes internode messaging through it used of an XOR metric to define the distance between points in key space, so yielding a topology of low diameter. Spearheaded by Maymounkov and David Mazières in 2002 [25], its distributed nature makes it highly resilient to the denial of service (DOS) attacks as the protocol would route across unavailable nodes. Further, its distributed nature provides no central point of failure.

In Ethereum, clients store information about peers in two data structures. The first is a long-term database, stored on a disk and persists across client's reboots. The database contains information about every node that the client has seen. Entries consists of a node Id, IP address, TCP port, UDP port, time of last ping sent to the node, time of last pong received from the node, and number of times the node failed to respond to a findnode message. If the time of last pong received from a node was older than one day, that node will be removed from the database. The second is a short-term database called routing (Kademlia) table and are always empty when the client restarts. The routing table represents a binary tree whose leaves are k-buckets. These k-buckets are a list of routing addresses of other nodes in the network, which are maintained by each node and contain the IP address, port, and nodeId of other participants in the network. The structure of the routing table is such that nodes maintain detail knowledge of addresses closest to them and exponentially decreasing the insight of more distant address space. The routing table for RLPx (*Kademlia-like table*) network consists of at least one and at most 255 rows. Each row contains

at most k peers or k-buckets where k defines the redundancy of the network. K is set to 16 for the node discovery, i.e., every k-bucket contains 16 node entries. Entries in the bucket are sorted by the time last seen, with most recently seen nodes at the tail and least seen at the head. Every peer in the network keeps a distance between 2^i and 2^{i+1} from itself. The distance between two arbitrary peers n1 and n2 is *distance (n1, n2) = keccak256(n1) XOR keccak256(n2)* [14]. Nodes in the P2P network are identified by a 256-bit cryptographic hash of the nodes' public key – public key serves as the node Id. The cryptographic hash is based on the elliptic curve secp256k1. A plausible reason for the 256-bit length is as simple as the fact that a key length of 256 provides plenty of space for a reasonable distance, whereas public keys are of unwieldy length. It could also allow the public key format to change independent of node discovery. It might be related to the EVM's native support for 256-bit integers. It could also have something to do with security, to avoid broadcasting public keys, for instance. Further, it could also be because the codomain for hashes are uniform whereas public keys are not. A designator for an Ethereum node aka enode is *<node ID>@<IP address>:<node port>*. According to Katkuri [20], we have the option of using separate ports for UDP node discovery and TCP P2P communication. In that case, the enode designator will be formatted as *<node ID>@<IP address>:<TCP port>?discport=<UDP Port>*.

3.2.2 Ping and Pong Packets

When a node n1 is discovered, Kademlia adds it to the corresponding bucket. However, if the corresponding k-bucket is full, the least recently seen node n2 in the k-bucket needs to be reevaluated by sending a ping packet, of packet-data = [version, from, to, expiration]. The fields are defined as:

- version: 4
- from: [sender_ip, sender_udp_port, sender_tcp_port]
- to: [recipient_ip, recipient_udp_port, 0]
- expiration: an absolute UNIX time stamp. Expired timestamps may not be processed

In response to a ping packet, a valid node replies with a pong packet with the following fields:

- to – which should be the same as the corresponding ping packet;
- ping_hash – hash of the corresponding ping packet;
- expiration – timestamp after which this packet becomes invalid.

Usually, nodes with pong packets corresponding to ping hashes received within 12 hours are considered valid. This ensure senders of a query participate in the discovery protocol to prevent DNS amplification attacks [14]. DNS amplification attack is a distributed denial of system (DDoS) that leverages DNS to overwhelm a victim server.

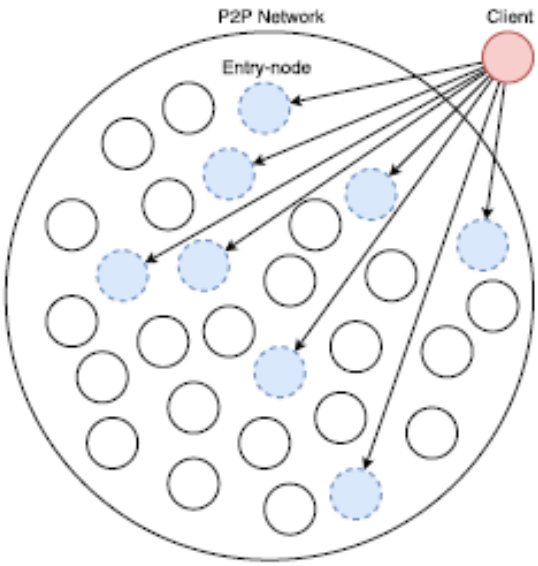

Fig3. Entry nodes (blue & dashed line) of a certain client (red) in the Ethereum P2P network (interconnections not depicted), arrows indicate an outgoing connection from the red client node

3.2.3 Find node and Neighbour Packets

Kademlia table initially maintains a set of bootstrap nodes used by a newly instantiated node to locate peers. Newly instantiated node (initiator) start by locating α (usually 3) nodes closest to the target. The initiator sends α concurrent *find node* and these α nodes respond with a list of k nodes from its own routing table that is closest to the target. Find node contains two fields: *expiration* and a *target* – 32-byte address, which does not have to be an actual node ID. The *neighbour* send list of nodes from its own table in response to the *find node* message. This list

of nodes [[ip, udp-port, tcp-port, node_id] ...] should be sent if the sender of the find node has been verified, to prevent amplification attack. The initiator (querier) adds discovered nodes to its routing table via this process and iteratively repeats the querying process until it converges on the target node. Packets are formatted as shown, where | | represents string concatenation:

```
packet = packet-header || packet-data
packet-header = hash || signature || packet-type
hash = keccak256(signature || packet-type || packet-data)
signature = sign (packet-type || packet-data)
```

The signature is an encoded byte array of length 65 of the concatenation of the signature values (random number, shared secret and the 'recovery id'). Packet-type is a single byte which defines the type of the message, 0x01 for ping, 0x02 for pong, 0x03 for find node and lastly 0x04 for neighbours.

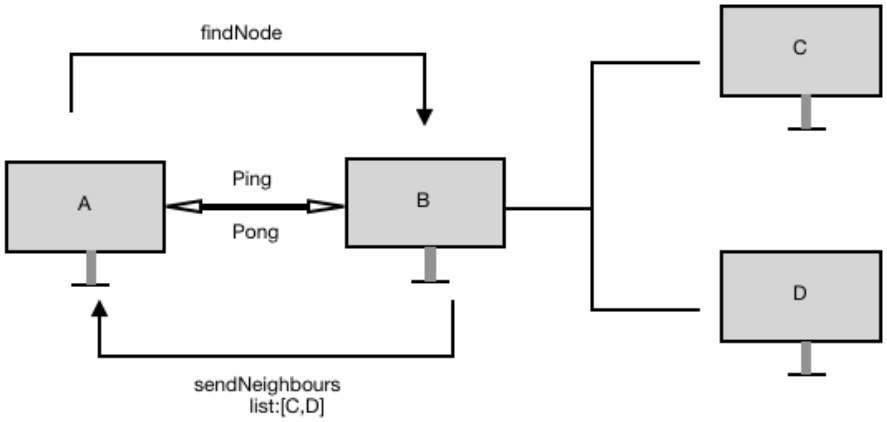

Fig4. Schematic view of the node discovery process. (courstesy of shyft media netwok)

3.2.4 Differences from standard Kademlia

RLPx node discovery differs from Kademlia node discovery in the following ways. First and foremost, RLPx does not support the store and find value operation in Kademlia, it only supports node discovery and routing. This prevents other nodes from directly interacting with the routing table. Second, RLPx uses 512-bit as the node Ids while Kademlia uses a randomly generated 160-bit. Third, RLPx does not calculate node distance directly on the XOR of nodes Ids. Instead, it calculates the distance between arbitrary nodes on the XOR of the kaccak256 hash of the node Ids. Lastly, nodes Ids are used to establish authenticated communications and are

signed with the Elliptic Curve Integrated Encryption Scheme (ECIES), maintaining integrity between nodes.
To summarize RLPx protocol suite, it contains at least two protocols:

- Kademlia for Ethereum node discovery that involves UDP and
- DEVp2p wire protocol for a node to node communication involves TCP packets

3.3 DEVp2p Wire Protocol

DEVp2p nodes communicate through the exchange of messages using RLPx. Nodes may listen and advertise on any TCP port or use a default port number of 30303 [15]. The nodes encompass TCP packets and further require a two-phase handshake to establish a secure session. In the first handshake, nodes exchange their public key (encrypted), which is used for subsequent communications. The connection is authenticated between the initiator and the receiver. In the second handshake, the nodes' exchange their capabilities, i.e., DEVp2p sub-protocol, version, and network IDs. ECIES is used in the RLPx handshake. According to Martínez et al. [16], the ECIES consist of the following key functional components:

i. a key agreement used for the generation of shared secrets among two parties;
ii. a key derivation which is a mechanism to produce a set of keys from keying material;
iii. hash digest;
iv. the symmetric encryption algorithm and
v. a message authentication code (MAC) used to authenticate messages.

Summarily, creating a secure session for P2P communication involves peer authentication and a base protocol handshake. The initiator and receiver (recipient) undergo the following steps for a secure connection:

i. Initiator connects to the receiver and sends an authenticated message;
ii. Receiver accepts, decrypts and verifies the authenticated message, send an acknowledgement and sets up a secure session;

iii Initiator checks authentication acknowledgment from a receiver derive secretes and establishes a secure connection;

iv Peers exchange their capabilities (base protocol handshake) on the established secure connection.

The DEVp2p sub-protocol messages are *hello (0x00), disconnect (0x01), ping (0x02)* and *pong (0x03)*. The second handshake is part of the *hello* message.

3.3.1 Hello Packet

This message is the first packet sent over the network and it is done once by both sides of the connection. No other message may be sent until the reception of a hello message. This message comprises:

i p2pVersion: specifies the implemented version of the P2P protocol (4 or 5 as of this writing);

ii clientId – specifies the client software identity as a human readable string (e.g., "Ethereum (++)/1.0.0");

iii capabilities – is the list of supported capabilities and their version;

iv listen_port – defaults to 30303. Zero means no peer is listening;

v node_id – secp256k1 encryption of a unique identifier for the node (public key).

3.3.2 Ping, Pong and Disconnect

The DEVp2p nodes periodically send a ping (keep-alive) message to ensure peer connectivity and the corresponding node should respond with a *pong*. However, if such pong messages are not forthcoming, the peer would receive a *disconnect message*, with a single byte reason code. According to Kim et al. [3], the majority of disconnects, both for sent and received messages in Ethereum major clients (i.e., Geth and Parity), is *too many peers* with a reason code of *0x04*.

3.4 Ethereum Sub-protocol

These subprotocols run on top of DEVp2p. They are self-contained and define sets of messages transported on the Ethereum Network over RLPx. In this report, focus

is on eth subprotocol version 62 and 63. Ethereum data such as blocks, headers, transactions are propagated through the ETH protocol, used by the sync loop of Ethereum clients to synchronize the chain. Other popular Ethereum sub-protocols include Light Ethereum sub-protocol (les), Swarm (bzz) and Whisper(shh) [17, 18, 19].

3.4.1 STATUS message

Subprotocol negotiation and initialization collect shared capabilities between the initiator and remote node. After the hello message, the first message communicated by both peers is the status message. This message also requires a two-way handshake. Both peers send status messages, and the Ethereum session is only active after the reception of status messages from either side of the network. Other messages may be sent afterward. This status message communicates an Ethereum peer current state. The fields for the status messages are:

i protocolVersion i.e., 62 or 63 for eth;
ii networks_id – main, test or private networks, etc;
iii total_difficulty (TD) – of the best chain (which is an integer found in the block header). The TD is the cumulative sum of the difficulties of all the blocks preceding the current block;
iv best_hash – hash of the best-known block (i.e., having the highest TD);
v genesis_hash: hash of the first block commonly referred to as the genesis block.

The fundamental requirements for a node to join a specific Ethereum network are to have the same genesis block and network id. A node will disconnect from an Ethereum network with the reason being useless peers (0x03) or incompatible protocol (0x06) if it has a different genesis hash or network id. After the status message, peers utilize the GenesisHash, TD and best_hash for syncing [20].

3.4.2 NewBlockHashes

Specify one or more new blocks which have appeared on the network. Nodes usually inform peers of blocks they are unaware of to reduce the bandwidth requirement of

the connection. A new block; [+0x07, [blockHeader, transactionList, uncleList], totalDifficulty] includes a TD needed to pick the best peer for synchronization. The calculated TD and the advertise TD are compared when the block is imported into the chain. The peer is dropped if the TD of the advertised peer does not match that of the calculated TD. Furthermore, a new block is also discarded if it is older than the uncle limit (7 blocks) or has exceeded 32 blocks. Each peer has a queue limit of 64 new blocks as an anti-DOS measure.

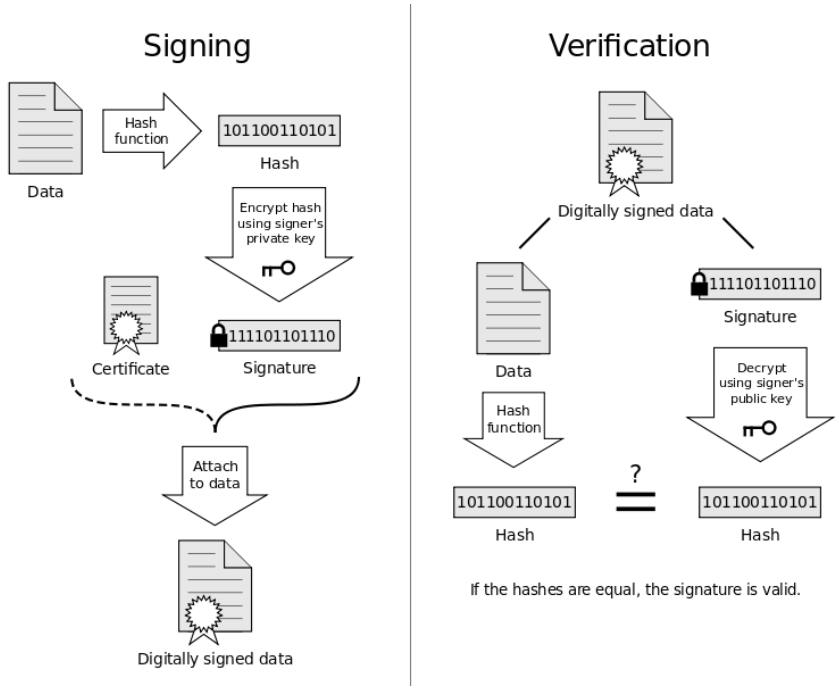

Fig5. Digital Signature Diagram (source: https://commons.wikimedia.org/wiki/File:Digital_Signature_diagram.su

3.4.3 Transactions

Use to propagate one or more transactions and nodes. Must resend at least one new transaction to a peer in the same session. A transaction field contains:

- a gas limit representing the maximum amount of computational step the transaction can take;
- a signature identifying the sender and proving the intent to send a message to the recipient in the blockchain;
- the recipient of the transaction and some optional data field which may contain the message to dispatch in the transaction;

- value field which represents the amount of Wei to transfer from the sender to the receiver;
- a gas price which is the amount of Wei paid per unit for the computational cost and a
- nonce.

Ethereum transactions use an account model, and the transactions are verified with an Elliptic Curve Digital Signature Algorithm (ECDSA) signature [36]. These verifications include making sure the account does not exceed the gas limits, having senders with enough Ether and transacting a positive value. Conceptually, there are two important aspects of an Ethereum account-model:

- Transactions are the state transition functions
- The results of these functions can be stored.

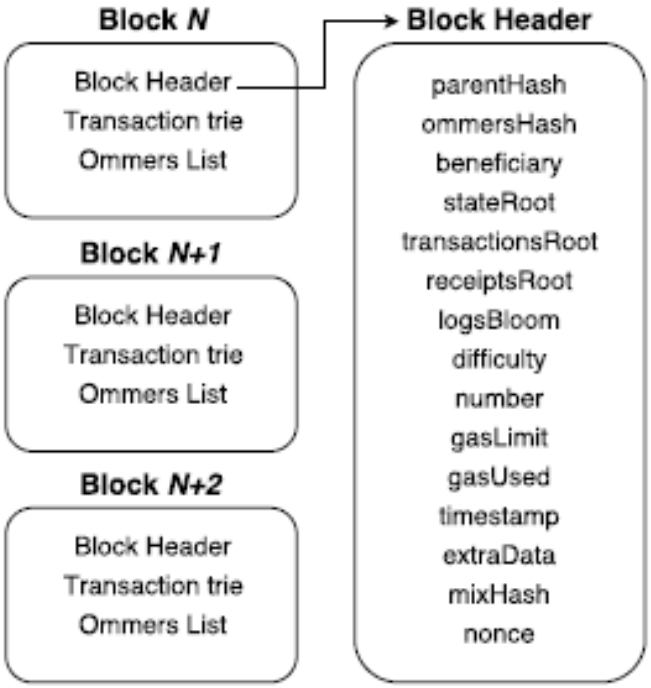

Fig6. Overview of the Ethereum blockchain and the data included in each block

A full or archival node typically stores the whole transactions and resulting state transitions for all block heights in a local data store. This includes all historical states, even those no longer valid. This typically allows potential clients to query the state of the blockchain at any time in the past without having to re-calculate everything from the beginning. This will likely require staggering amounts of disk

storage and because it is not strictly necessary, conceptually blockchain data can be carefully separated:

- chain data (the list of blocks producing the chain)
- state data (the result of each transaction's state transition)

While all chain data is needed to ensure the cryptographic chain-of-custody and that nothing has been tampered with, old state data can be discarded. This process of reducing the space requirement is known as pruning. This is because state data represent implicit data, i.e., its value is known merely from calculations, not from the genuine information communicated. By contrast, chain data is explicit and stored as the blockchain itself. Inasmuch, while both chain and state data are stored locally on the node's disk, only the chain data is strictly necessary. The state data can be ephemeral.

3.4.4 GetBlockHeaders and BlockHeader

These are used to download a local copy of a full blockchain. A GetBlockHeader request returns a BlockHeader. A BlockHeader is an RLP-encoded list of block headers where each header has the following fields; *parent_hash, uncles_hash, coin_base, state_root, receipt_root, transaction_root, bloom, difficulty, block_number, gas_limit, gas_used, timestamp, extra_data, mix_hash* and a *nonce* (TD) [27].

3.4.5 GetBlockBodies

They are used to retrieve full block contents and verify the validity of blockchain. There are two types of blockchain validation: block header and blockchain state validation or full node validation. Block header validation checks the Ethash, gas limit, nonce, block parent hash, block number, and timestamps. For full node validations, the node will verify the legitimacy of the blockchain by replaying all transactions from the genesis block. Once this is completed, the node is sync and officially a full node. You do not need to run an archived Ethereum node for blockchain state validation. Blockchain state validation requires more computational power and storage space but the usage of the Merkle-Patricia state tree immensely reduce the storage requirements [6].

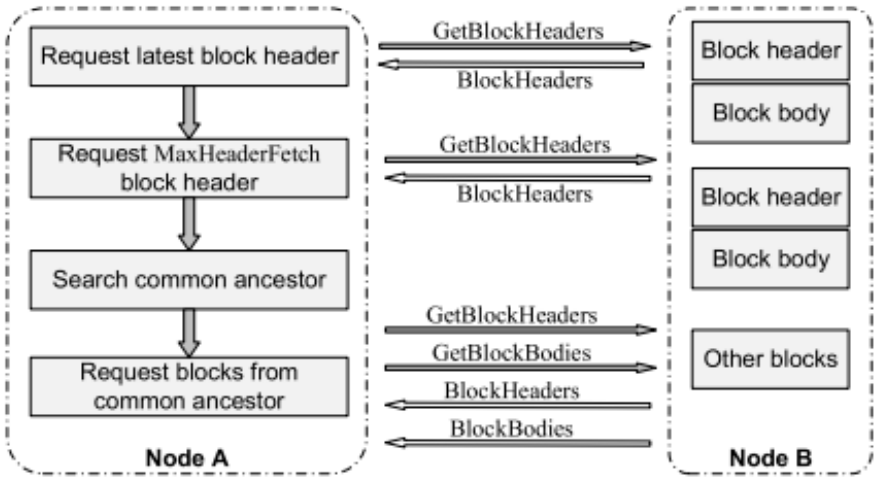

Fig7. Block synchronization process between nodes (source [28])

For a regular synchronization between two nodes, the process is as follows:

i Node A having a lower total difficulty can request for the header of the latest block from node B (higher total difficulty). This is implemented by sending a GetBlockHeaders message. B will reply with a BlockHeaders message containing the specified block header requested by the GetBlockHeaders message.

ii A request a MaxHeaderFetch (=256) blocks to find the common ancestor from node B. B sends up to MaxHeaderFetch but may send fewer.

iii If A has not found common ancestor after the above two steps, node A will continue to send GetBlockHeaders message, requesting one block header each time. Moreover, A repeats in binary search to find the common ancestor in its local blockchain.

iv As soon as A discovers a common ancestor, it will request block synchronization from the common ancestor. In this process, A requests MaxHeaderFetch blocks per request, but B may send fewer.

3.4.6 GetNodeData and GetReceipts

Ethereum fast synchronization (eth63) makes use of GetNodeData to download a global state database at that node, and GetReceipts to store the state after a transaction has been executed and are placed in an index-keyed trie. Transaction receipts contains: gas consumption; logs; bloom filter created based on the information found

in the logs; a post-transaction state or status code encoded as a byte array. The goal of the fast sync algorithm is to exchange processing power for bandwidth usage. Instead of processing the entire block-chain one link at a time and replay all transactions that ever happened in history, fast syncing downloads the transaction receipts along the blocks. It subsequently pulls an entire recent state database [26]. This allows a fast-synced node to nevertheless retain its status and archive node containing all historical data for user queries (and therefore not influence the network's health in general). This enables fast sync to reassemble a recent network state at a fraction of the time it would take for full block processing. An outline of the fast sync algorithm would be:

- Similarly, to classical sync, download the block headers and bodies that make up the blockchain;
- Verify the header chain's consistency (POW, total difficulty, etc);
- Instead of processing the blocks, download the transaction receipts as defined by the header;
- Store the downloaded blockchain, along with the receipt chain, enabling all historical queries'
- When the chain reaches a recent enough state, pause for state sync:
  - Retrieve the entire Merkel Patricia state trie defined by the root hash of the pivot point;
  - For every account found in the trie, retrieve its contract code and internal storage state trie;
- Upon successful trie download, mark the pivot point as the current head;
- Import all remaining blocks by fully processing them as in the classical sync.

By downloading and verifying the entire header chain, we can guarantee with all the security of the classical sync, that the hashes (receipts, state tries, etc) contained within the headers are valid. Based on those hashes, we could confidently download transaction receipts and the entire state trie afterward. Additionally, by placing the pivoting point (where fast sync switches to block processing) a bit below the current head. This ensures even larger chain reorganizations can be handled without the need of new sync (as we retain all the state going that many blocks back). This

fast sync algorithm requires the functionality defined by eth/63. Note: if sync fails because of header verification the last **N** headers must be discarded as they cannot be trusted enough.

3.5 Consensus in Ethereum

Blockchain-based technologies like Ethereum delivers the opportunity for universal participation and typically used incentivization to recruit participants to its network. These incentives are lucrative and hence competition among participants is so fierce to confirm the next block on the chain. In this manner, we need a consensus protocol to confirm the next block and distribute the rewards accordingly. Consensus algorithms are one of the most revolutionary aspects of this blockchain technology. Without this consensus, blockchain technology would have been architecturally easy to develop. They are needed to establish an irrefutable system of agreement between various nodes across a distributed network marred with malicious users. That is where the genuine power of blockchain is built - decentralization. Sustaining a democracy around the data that is being built, used and shared. Consequently, every node which possesses a copy of this ledger, having to have that consensus algorithm to be able to verify that transactions are real and accurate before they are appended to the ledger. The Ethereum protocol must achieve consensus in the face of two types of obstacles: imperfections in the network, such as high latency and nodes crashing, as well as deliberate attempts by some nodes to subvert the process. According to Azouvi et al. [30], the consensus algorithms for blockchains must satisfy three security properties: liveness, fairness, and unpredictability. By liveness, we mean the chain inevitably grows. Fairness implies uniformity in electing a new block even in the presence of malicious nodes and finally, unpredictability means an adversary is unaware of the eligibility of the block ahead of time. Blockchain consensus deviates from classical distributed systems consensus algorithms like PAXOS and FLP through it used of incentives to coerce nodes to behave appropriately. Further, Ethereum nodes do not have persistent or long-term identities. One reason for this lack of identities is that in P2P system, there is no fixed authority to assign identities to participants to verify there are not creating nodes at will. The technical term for this is sybil attack. The other reason is that

pseudonymity is inherently a goal of Ethereum [27]. Ethereum currently employs the proof of work consensus algorithm, and plans are made to move to a proof of stake.

3.5.1 Proof of Work

In Proof of Work (PoW), participating nodes work to unravel complex mathematical puzzle which usurps a lot of computational power to confirm the next block on the chain. The chief operating principles are a complex mathematical puzzle and the ability to trivially verify the solution across the network. Nodes working to confirm the next block in the chain are known as miners and the process is called mining. Mining serves two purposes. It is wielded to verify the legitimacy of a transaction, i.e., avoiding the so-called double-spending attack; and creating new digital currencies by rewarding miners for performing the previous work [32, 33, 34]. The mining process is an operation of inverse hashing: it determines a number (nonce), so the cryptographic hash algorithm of block data results in less than a given threshold - difficulty level [29]. Producing a nonce is usually a random process averaging lots of trials and error. To emphasize, protection against an invalid transaction is completely based on cryptography which is enforced by consensus. This implies if a node attempt to include a cryptographically invalid transaction, the sole reason the transaction won't end up in the long-term consensus is that a majority of the nodes are honest and won't include an invalid transaction in the chain. Contrarily, protection against double-spending is solely on consensus. Two transactions representing a double-spend attempt are both valid from a cryptographic perspective. But it is the consensus that determines which one will end up in the long-term chain.

Ethereum PoW is called EtHash and uses Keccack – a hash function eventually standardized to SHA-3 [27]. The mining difficulty is adjusted to around 15 seconds. EtHash emphasizes on a property termed *memory hardness*, to combat mining centralization. Memory hardness means performance is limited by how fast your computer can move data around in memory rather than how fast it can perform mathematical operations [10]. This technique inhibits the use of Application Specific Integrated Circuits (ASICs) where a disproportionately compact group of miners could enjoy autonomy over the network. EtHash equally implements the

GHOST protocol to combat centralization of mining power. GHOST includes the headers of recently staled blocks – or Orphans as Ethereum designates them. The nodes producing the uncle block, and the node including it in the blockchain are given a reduced value. The reward includes stale of 87.5% - the nephew also receives 12.5% of the block reward. This encourages them to continue with the recent block in the blockchain [29].

Newly minted blocks are spread among the miners over a P2P structure, with each miner keeping the lengthiest chain as the winner [31]. Since having the longest chain entails unraveling the cryptographic puzzle of all preceding blocks, overturning a long tail segment is hard. For subverting consensus in the network, a 51% attack is needed [29]. Ethereum developers are working to move the PoW algorithm to a proof of stake algorithm.

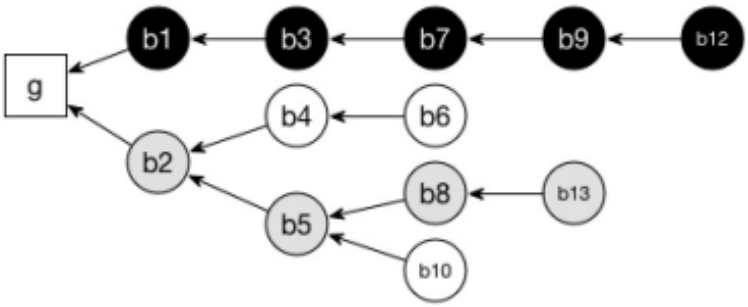

Fig8. Nakamoto's consensus protocol at the heart of Bitcoin selects the main branch as the deepest branch (in black) whereas the Ghost consensus protocol at the heart of Ethereum follows the heaviest subtree (in gray).

3.5.2 Proof of Stake

Unlike PoW where participating nodes solves mathematical puzzles which usurp lots of computational power, proof-of-stake (PoS) is about validating and it aimed to avoid depletion of computer resources and centralization of mining power. Ethereum PoS (Casper) provides the block-inclusion decision making power to those entities that have a stake in the system. PoS requires the stakes to be pre-distributed at the beginning of the process which was not the case with the PoW approach. Hence, the stakeholder's chances of extending blockchain by including its own block depends proportionately on the number of stakes it has in the system [34, 35]. From an

academic perspective, there are two flavours of PoS: Chain-based and Byzantine Fault Tolerance (BFT) style. Chain-based PoS relies on the synchronicity of the network whilst BFT-style favour consistency of nodes over availability. In [70], the 51% attack on the Casper PoS are:

i Finality reversion: validators that already finalized block A then finalize some competing block A', thereby breaking the blockchain finality guarantee;

ii Liveness denial: validators stop finalizing blocks;

iii Invalid chain finalization: validators finalize an invalid (or unavailable) block;

iv Censorship: validators block some or all transactions or blocks from entering the chain.

3.6 Sharding Ethereum Blockchain

As earlier noted, nodes are responsible for verifying a block and making sure the consensus rules are followed. In so doing, the best way is to preserve a full copy of the public ledger, thereby making it easier to verify a miner work. Accordingly, every node processes every transaction and every node store all the state. The Ethereum blockchain is approaching 1TB [68] so it becomes impossible for an ordinary computer to manage a full node. This begs the question, if Ethereum nodes become too expensive to operate, then the network will be more susceptible to centralization. Requiring every transaction to be verified by all nodes will make scaling of the Ethereum node impractical. If a blockchain is both scalable and secure, it means our chain is centralized and that we have higher throughput. Right now, Ethereum is a decentralized public ledger. How do we break this trilemma to include scalability in the current model? Ethereum 2.0 introduces the concepts of sharding [23] to solve the scalability trilemma of decentralization, scalability, and security. Sharding refers to splitting the entire blockchain into groups called shard. Each shard will contain its unique sets of account balances and/or smart contracts. Accounts can merely send transactions to or call accounts in the same shard. Each shard gets its own set of validators (therefore, PoS is a pre-requisite), and these validators will not normally need to validate all the shards. It is meaningful to note the following

vocabulary before we dive into sharding:

- State: the unified sets of information that describe a system in a specific time.
- Transaction: an operation issued by a user that changes the unified state of the system.
- Merkle root: a data structure accumulating substantial data via a cryptographic hash. It is straightforward to check if a piece of data is part of the structure in a very short period.
- Receipt: a side-effect of transaction that is not stored in the state of the system. This is preserved in the Merkle root, so its existence can be verified easily. For instance, smart contracts log in Ethereum are kept as receipts in the Merkle root.

Transactions between accounts of the same shard would work in the same way as they work today. Transactions wishing to communicate across multiple shards will need to employ some special techniques, based on the concept of transaction receipts. The crucial difference between calling a contract directly and verifying the receipt is that for direct call one needs to execute the code of the contract you're calling. But for verifying a receipt you nothing but a need to be sure that receipt cannot be produced by anything else than the transaction you want. For instance, if you ideally want to accept payment in tokens managed by a different shard, you would generate the payment ID.

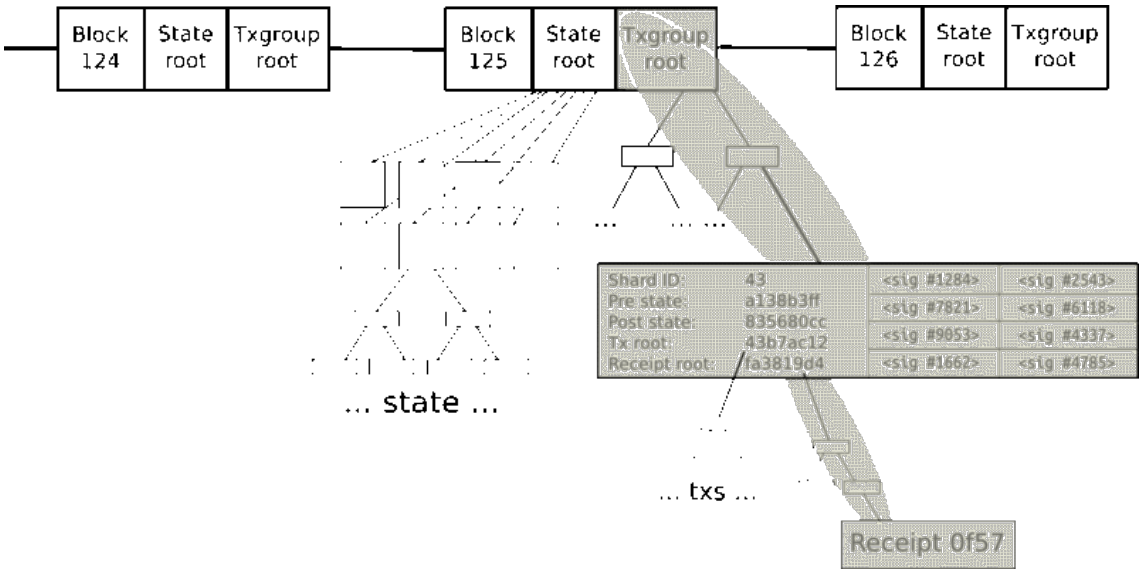

Fig9. Ethereum choose to follow the receipt paradigm for cross-shard communications. Transaction is valid if: pre-state root equals shard root in global state; Signatures (2/3 of randomly pre-selected set of 135) are valid. (Image courtesy of hackernoon)

You then need to give it to the payer, invite the payer to pay in the remote shard (with payment ID), and 'bring you back' the receipt. Sharding allows scaling Ethereum further because not all nodes of the network will have to execute all the transactions. It additionally allows for higher transaction throughput. The requirement is to make sure the appropriate shard obtains the transaction within an appropriate propagation delay. If we take a group of 100 validators/shard with each one connected to 10 other validators. You can, therefore, be 2 hops away from every validator in your shard. Transactions can come from either other validators or someone from outside your validator cluster. If we further assume our node is connected to 100 other nodes. Even with a huge overlap — say, essentially, you are only connected to 20 distinct nodes. If there is one node per person, we can make some assumptions about efficient routing and find you'd never have more than 8 hops away from the required validator cluster. If we have an established set of validators, you can directly connect to one of the ones involved in your transaction, as with a client-server architecture.

CHAPTER 4

The EthereumJ Client

To gain access to the Ethereum Network, we must initially connect to an Ethereum node i.e. a peer running the official client. An Ethereum client is a software application that implements the Ethereum formal specification and communicates over the P2P network with other Ethereum clients. Ethereum allows interoperability between different clients if they comply with the reference specification and standardized communication protocols. These reference specifications are contained in the "Yellow Paper" [27]. The paper is periodically updated as major changes are made to Ethereum. The paper, in addition to various Ethereum Improvement Proposals (EIP) [83], defines the standard behaviour of an Ethereum client. As a result of this clear formal specifications, they are several independently developed, yet interoperable software implementations of Ethereum clients. Whilst the multiple Ethereum clients are implemented by separate teams and usually in different programming languages, they all "speak" the same protocol and follow the corresponding sets of rules. The broader diversity of Ethereum clients is typically regarded as good for the network. Indeed, it has, for example, proven itself to be an excellent way of defending against attacks on the network [22]. This is so because the exploitation of a specific client's implementation strategy simply hassles the developers while they patch the exploit, at the same time as other clients keep the network running almost unaffected. They are currently six main implementations of the Ethereum protocol written in six different languages: Geth, written in Golang; Parity written in Rust;

Mantis, written in Scala; Harmony and EthereumJ, written in Java; cpp-ethereum, written in C++ and; pyethereum, written in python [71].

4.1 Configuration Properties

The EthereumJ configuration file defines several default properties for our client:

- A list of seed pairs (25) to initiate the search for online peers. Five Geth, 19 Parity, and one C++ Ethereum discovery node. These nodes are known as the bootstrap nodes. It is just a list of nodes to start and jump into the network as fast as possible for a new peer. They are usually alive and maintained by the Ethereum development team;
- A default port (30303) to listen for incoming connection;
- A list of trusted peers. Trusted peers always accept an inbound connection. The EthereumJ client sets this list to empty as default;
- The list of protocols supported by peers is eth, bzz and ssh. The default protocol is set to eth v63. However, there is an option to use v62 – does not support fast synchronization;
- A maximum "active peer" list of size 30. Extra peers connecting to us will be dropped with *too many peers* message. However, peers from the "trusted list" are always accepted. Active peer list is a terminology used to describe peers that are actively synchronizing the chain. EthereumJ implemented the term "Other Connected peers" to describe peers which are listing for new blocks and transactions;
- The peer discovery mode set to true. This allows the discovery protocol to find other nodes and connect to them. If the peer discovery is off, this peer will only be connecting to the peers from the active list (boot node list). Also, inbound connections from other nodes will be accepted.
- The discovery protocol reconnects to successful nodes after every 600 seconds and the maximum number of nodes to reconnect being set to 100;
- Discovered nodes and their established reputations are stored in the local database (rocksdb) and will be persisted during a virtual machine (VM) restart. The default blockchain is blockchain.config.name = "main" and it enables pruning to preserved disk space;

- connection.timeout of two seconds and channel.read.timeout = 30, i.e., how much time to wait for a message to arrive before terminating the connection. Blockchain synchronization is set to true and a minimum of 30 peers to be used for the sync process. Furthermore, it sets the network Id to 1 (Frontier-Homestead). Other network options are 2 - Ropsten test network, 3 – Ethercamp test network, and 4 - pre-Frontier Olympic network);
- writeCacheSize = 64, size in Mbytes of the write buffer for all datasources (state, blocks, transactions). Data is flushed to DB when write caches size exceeds this limit;
- stateCacheSize = 384, total size in Mbytes of the state DB read cache;
- headerQueueSize = 8, the size of header queue cache during import in Mbytes;
- maxStateBloomSize = 128, maximum size (in Mb) the state bloom fiter can grow up to.
- blockQueueSize = 32, the size of block queue cache to be imported in MBytes
- p2p.version = 5, framing.maxsize = 32768, eip-8 = true, genesis = frontier.json. Some of the default values in frontier.json are:
  - "nonce": "0x0000000000000042",
  - "difficulty": "0x0400000000",
  - "mixhash":"0x0000000000000000000000000000000000000000000000000000000000000000",
  - "coinbase": "0x0000000000000000000000000000000000000000",
  - "timestamp": "0x00",
  - "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000
  - "extraData": "0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa
  - "gasLimit": "0x1388".
  - genesisHash: d4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
- The pivoting hash when fast sync is enabled is:
  - pivotBlockHash=6149ddfd7f52b2aa34a65b15ae117c269b5ff2dc58aa839dd015790553269411
    i.e., where fast sync switches to block processing. Pivot distance from head is 1024.

4.1.1 Kademlia Routing

The Kademlia protocol maintains a routing table for node discovery. Kademlia for EthereumJ demarcates 256 rows and bucket size of 16. It also marks out a concurrency variable (α) of three. It further defines a request timeout, bucket refresh interval and a discovery cycle of 300ms, 7200ms and 30s respectively.

4.1.2 Peer Server

The *org.ethereum.net.server.PeerServer* and *org.ethereum.net.client.PeerClient* class both utilize the Netty framework. Netty is an asynchronous event-driven network framework for rapid development of maintainable high protocol servers and clients [72]. The *PeerServer* class defines two *EventLoopGroup* – one for the 'boss' (*bossGroup*) and the other for the 'worker' (*workerGroup*) since we are implementing a server-side channel. The *EventLoopGroup* manages how many threads are created and how they are assigned to the channels. The 'boss' accepts incoming connections and the 'worker' handles the traffic for the accepted connection. The 'boss' and 'worker' groups are being assigned to a *NioEventLoopGroup instance* – is a multithreaded event loop that handles I/O operations. The *PeerServer* also adopts a *ServerBootstrap* class to set up a server. Furthermore, a *NioServerSocketChannel* class is used to instantiate a new channel to accept incoming connections. The options specified for the incoming connections are keepalive, a message size estimator (maximum size of the packets - 1024bytes) and the connection timeout – in milliseconds. To configure a new channel, the *ChannelInitializer* handler is overwritten by the *EthereumChannelInitializer* class. This class declares the method for detecting and removing bad addresses, maintaining the queue threshold, preventing channel abuse, avoiding too frequent connection attempts, and lastly, it overrides the *initChannel* method from the *ChannelInitiazer* handler – it is a special handler that helps a user configure a new channel. After all the nitty-gritty, a *ChannelFuture* object is used to bind a port for syncing. We can then call the bind method with as many addresses as we choose – default bind port is 30303.

4.1.3 Peer Client

This class defines the client side for the Ethereum network. This class adopts a single *EventLoopGroup* which is used as both a 'boss' and 'worker' group. Furthermore, it made use of a Bootstrap class to set up a client-side channel. Unlike the *PeerServer* class, the *NioSocketChannel* is being used to create a client-side channel. The client does not have the *childOption* since the *SocketChannel* does not have a parent. Like the PeerServer channel-handler, the *PeerClient* channel-handler is also overwritten by the *EthereumChannelInitializer* object and uses the connect method to establish a link with the *PeerServer*.

4.2 Discovery Activities

Our client initiates a connection to a remote address and establishes a listener for incoming connections respectively using the *NioSocketChannel* and the *NioServer-SocketChannel* classes from the Netty framework. EthereumJ makes use of the spring framework for dependency injection. The library defines an org.ethereum.net.rlpx.discover package responsible for node discovery activities. A node in our network can be in one of the following states:

- Discovered: the current node was discovered either by getting it with a neighbour message or by receiving ping from a new node. In either case, we are sending a ping and expecting a pong. If the pong is received within the specified message interval of 1500ms, the remote node enters the 'alive' state otherwise the 'dead' state;
- Dead: the node didn't send the pong message within the specified timeout. This is the final state;
- Alive: the node responded with pong and it's now a candidate for inclusion to the routing table. If the table has bucket space for this node, it is added to the table and it becomes active. Assuming the table bucket is full, the new node must challenge an old node from the bucket. If it wins, the old node is relinquished, with the new node added to the bucket (active) and if it loses the challenge it becomes nonactive;
- Active: the node is in the bucket;

- EvictCandidates: this node is in the bucket, but it is currently challenging with a new node to retain its position in the table bucket;
- NonActive: recognized as the veteran state. These are nodes which were alive and even active but now retired due to losing to a challenge. It is an option to return veterans back to the table bucket.

The pseudocode of the discover task responsible for sending a find node message is shown in snippet 1. The RLPx node discovery protocol first attempt to clinch closest nodes to the node id. Closest nodes are by a measure of the XOR distance of the Kaccak256 hash of the node ids. The maximum allowed size of closest nodes returned is 16 – the defined Kademlia bucket size. Next, we iterate through all the nodes in the *closest_nodes* list, sending a find node message to each node not previously seen. This discovery task represents a recursive process that terminates after a given number of rounds (eight by default), or if the list of seen nodes after each round is empty. Two scheduled executors are used for the discover task:

- The first executor schedules a discovery task every 30 seconds (Kademlia discovery cycle).
- The second executor sets up a refresh task every 7200ms (Kademlia refresh bucket). The refresh task calls the discover task.

A UDP listener later starts the scheduled executor service responsible for the discover tasks. The listener uses the multithreaded NioEventLoopGroup responsible for I/O connections as specified by the Netty framework. On initial connection initiation, we create a binding by port and IP address to the bootstrap nodes and subsequent nodes which appear on our network through the discovery activities.

```
1. method discover (nodeId, round, previousTried):
2.     if round equals to max step (8 by default), return
3.     define closest_nodes to be the list object of nodes closest to nodeId;
           //closest nodes are by a measure of the XOR of the kaccack256 hash between node ids
4.     define tried_nodes as a list object of nodes // tried_nodes initialized to zero
5.     for all nodes in closest_nodes
6.         if node_x is not in tried_nodes and not in previousTried list
7.             send a find node to node_x
8.             add node_x to tried_nodes list
9.             if tried_nodes size equals to 3 (concurrency parameter, alpha)
10.                break
11.    if tried_nodes is empty
12.        return
13.    add all previousTried to the tried_nodes list
14.    discover (node id, round+1, tried_nodes) // make a recursive call to the discover method
```

Snippet 1: Pseudocode of the node discovery activity. For each node closest to the node id, the discover task sends a find node message to get even more nearby nodes.

4.2.1 Synchronization Pool

For managing peers involve in blockchain synchronization, an initialization method is defined to contain a set of tasks that executes every 3 seconds. These tasks are:

i An update of the local blockchain difficulty to include the difficulty of the most recent block. The difficulty of a block is a measure of the amount of computation effort needed to mine the block. Blockchain difficulty is the culminative sum of all the blocks in the chain.

ii Preparation of the active peers from the TCP connection. This step is meant to filter peers in the synchronization pool, committing only peers that are actively in synchronization with the local blockchain. Regarding this, the first order of business is to test all peers in the 'active list' –

specified in config file, since server connection peers were untested. We denote this test in snippet 2 as ***filtrationTest*** and they include:

- Checking if the reputation of the remote node is predefined (default value – 1000500). The EthereumJ client assigns a predefined reputation value to each node in the network on first contact. This value may increase or decrease depending on whether the remote node responds to a ping, pong, neighbor, handshake, authentication or disconnection messages;
- Return false if the reputation value of the remote node is less than 100
- Return false if the remote node is not valid i.e., not on same network id, genesis hash, wrong fork, etc.
- Return true if the remote blockchain difficulty is at least equal to the local blockchain difficulty.

Peers who succeed this ***filtrationTest*** are added to a local list – ***localActive***. From this list, a 20 percentile of peers with highest blockchain difficulty is realized and stored in a filtered list – ***filteredList***. A 20 percentiles by highest blockchain difficulty is assumed since nodes with these difficulties have most likely seen the most recent block. A variable cap whose value is dependent upon the synchronization status of the local blockchain is defined. The number of connections to kill is indicated as ***killCount*** = ***localActive*** – ***filteredList*** – ***cap***. The second order of business is to drop all the server connected idle peers which are not in the 20 percentiles. These drops are done with a reason message of "too many peers. All the filtered peers which are not in the synchronization pool are then added to it. The pseudocode for the preparation of the active channel is shown in snippet 2.

iii Filling up the active channel. This is necessary if the count of the connected peers is not up to the default (30) 'max active list' – specified in the config file. We need to query the Kademlia table for the top performing nodes not already in the channel. Top performing nodes are measured in terms of blockchain difficulty and reputation values. The number of nodes

retrieve from the table is at most *killCount*. A connection is made to all the retrieved nodes from the table.

iv Cleaning up the channel, i.e., removing peers from the channel due to a disconnection

```
1.method prepareActive ():
2.      for all peers in channel
3.         if filtrationTest
4.                  add peers to localActive list
5.      if localActive list is empty, return
6.      threshold = min (synchronization peer count, localActive)
7.      compute 20PercentRange (20 percentiles of the top peers by blockchain difficulty)
8.      while threshold >= 0
9.              if 20PercentRange
10.                 update threshold and break
11.     defined filteredList to be the sub list of localActive and threshold size
12.     if synchronization is complete
13.             cap = max (maxActivePeers/2, maxActivePeers - 10)
14.             // Ten peers are usually enough on a variance in data on short synchronization
15.     else
16.             cap = maxActivePeers/6
17.     killCount = localActive - filteredList - cap
18.     if killCount > 0
19.             for all peers in channel
20.                 if peer not in filteredList
21.                     if channel is idle
22.                         drop connection with reason too many peers
23.                         if dropped connection greater than or equal to killCount, break
24.     for all peers in filteredList
25.        if peer not in 'active peers' list //active peers are those in sync with our blockchain
26.                 add peer to the synchronization pool
27.     clear the active peer list
28.     add all peers in the filteredList to the active peers list
```

Snippet 2: Pseudo code for preparing the active peers in the channel.

4.3 Node Disconnections

According to the EthereumJ client source code, the disconnection messages are defined as follows:

- REQUESTED (0x00) – Disconnect requested by other peers;
- TCP_ERROR(0x01) – TCP sub-system error;
- BAD_PROTOCOL(0x02) – Packet cannot be parsed e.g., a malformed message;
- USELESS_PEER(0x03) – the peer is unwilling or delivers unreliable data;
- TOO_MANY_PEER(0x04) – Already too many connections with other peers;
- DUPLICATE_PEER(0x05) – At present a running connection with this peer;
- INCOMPACTIBLE_PROTOCOL(0x06) – Version of the P2P protocol is not the same as ours. For example, a client on eth will be disconnected from a client running les, shh or bzz protocol. Also, the client running on a different network Id would be disconnected;
- NULL_IDENTITY(0x07) – Null node identity received - this is automatically invalid;
- PEER_QUITING(0x08) – peer quits voluntarily;
- UNEXPECTED_IDENTITY(0x09) – i.e. a different identity to a previous connection/what a trusted peer told us;
- LOCAL_IDENTITY(0x0A) – Identity is the same as this node (i.e. connected to itself);
- PING_TIMEOUT(0x0B) – Timeout on receiving a message (i.e. nothing received since sending the previous ping);
- USER_REASON(0x10) – Some other reason specific to a subprotocol;
- UNKNOWN(0xFF) – Reason not specified.

Sometimes, disconnections message of the type requested and too many peers seem ambiguous for messages received. A peer P connected to a peer Q may receive a requested message because P needs to free one of its slots, and Q receives this because it is in one of such slots. A *requested* message could also be because P requires different protocol standards not fulfill by Q. Instead, *too many peer* messages are routinely sent on the connection attempt.

Nodes in the Ethereum P2P Network assign their peers a reputation value. This value is updated to maintain track of how well they behaved in the past and evaluate

whether it would be preferable to disconnect from them. The pseudocode in *snippet 3* shows the test necessary to establish ascertain a connection to a remote node is not a dead end. The *inbound_connection_bad_timeout* is set to two minutes by default with a list size of 500 to record peers that misbehave.

```
1. method isReputationPenalized ():
2.      if remote node is on wrong fork, return true;
3.      if node was disconnected && last remote disconnection reason is too many peers &&
                last disconnection time < inbound_connection_bad_timeout;
4.                      return true
5.      if last local disconnect equals null identity, return true
6.      if last remote disconnect equals null identity, return true
7.      if last local disconnect equals incompatible protocol, return true
8.      if last remote disconnect equals incompatible protocol, return true
9.      if last local disconnect equals useless peer, return true
10.     if last remote disconnect equals useless peer, return true
11.     if last local disconnect equals bad protocol, return true
12.     if last remote disconnect equals bad protocol, return true
```

Snippet 3. Pseudocode to check if a peer reputation is penalized

Some of the default value concerning reputation values are; a predefined – 1000500, handshake – 3000, authentication message – 1000, discover ping – 1. Nodes in our synchronization pool need to have a certain reputation value to maintain its place. We know from previous discussions (chapter 4.2.1) that this reputation value should be at least 100. Normally speaking, nodes with higher reputations are arguably at a more considerable advantage of being chosen for a block information as compare to less reputable ones. Each TCP session has a reputation system which either increases or decreases base on successful connection and failed connections respectively. The pseudocode in snippet 4 illustrate how to compute a session reputation. There are two leading factors in a session reputation; the rlpx reputation which value depends on the authentication and handshake messages count and; the discover reputation with value influenced by the ping, pong and neighbours messages count. This session fair reputation is then used together with other metric as illustration in *snippet 6*

to compute the reputation of a given node.

```
1. method sessionFairReputation ():
2.      if the number of discovery outgoing ping message equals the incoming pong message
3.              add min (outgoing pong, 10) * 2 to the discover reputation
4.      else
5.              add min (outgoing pong, 10) to the discover reputation
6.      add min (received neighbor messages, 10) * 2 to the discover reputation
7.      if rlpx sent authentication message is greater than zero
8.              add 10 to the rlpx reputation
9.      if rlpx handshake is greater than zero
10.             add 20 to the rlpx reputation
11.     add min (rlpx in messages, 10) * 3 to the rlpx reputation
12.     if node was disconnected
13.             if last local and remote disconnection reason is null
14.                     multiply the rlpx reputation by 0.3
15.             else if last local disconnection reason not equal to requested
16.                     if last remote disconnection reason equals too many peers
17.                             multiply the rlpx reputation by 0.3
18.                     else if last remote disconnection reason not equal to requested
19.                             multiply the rlpx reputation by 0.2
20.     sessionFairReputation = discover reputation + 100 * rlpx reputation
21.     return sessionFairReputation
```

Snippet 4: Pseudocode for computing a session fair reputation

```
1. method getSessionReputation ():
2.      if reputation is predefined
3.              return sessionFairReputation + predefined reputation
4.      else
6.              return sessionFairReputation
```

Snippet 5. Pseudocode for computing a session reputation

```
1. method getReputation ():
2.      if isReputationPenalized()
3.              return 0;
4.      else
5.              return persistedReputation/2 + getSessionReputation()
```

Snippet 6. Pseudocode for computing a node reputation. Persisted Reputation is initialized to zero on first contact with a node.

CHAPTER 5

Implementation

To attain a deeper understanding of the Ethereum Network, an EthereumJ node was deployed on the Mainnet network (genesisBlock: *d4e56740f876aef8c010. . . . . 0db1cb8fa3* and network Id = 1), at 10 pm on March 18^{th} – 10 pm March 22^{nd}, 2019. Our implementation is based on an Ubuntu LTS 16 laptop, Intel core i7, 1TB SSD, 16GB RAM, 8 CPUs, 103.5 Mbit/s download speed and 91.64 Mbit/s upload speed. As noted in [68], the Ethereum blockchain is growing at a rapid rate. Allowing the default synchronization (full sync) of EthereumJ would imply days or even weeks to achieve full synchronization. Therefore, fast synchronization was enabled, and the node was synchronized after 12.5 hours (figure 10). As discussed in chapter 3.4.6, instead of processing the entire blockchain one link at a time and replay all the transactions that ever happened in history, fast synching downloads the transaction receipts along the blocks, and pulls an entire recent state database. In addition to the fast sync, the default values of writeCacheSize, stateCacheSize, blockQueueSize, maxBloomSize detailed in chapter 4.1 was changed to 1024, 1024, 64, and 1024 respectively. These adjustments were made on the configuration file with the host machine specification in mind. Apart from these changes, all the default configuration properties described in chapter 4.1 were maintained.

We analyze the messages sent and received for node discovery, DEVp2p, and the Ethereum sub-protocol layers. For debugging and data collection purposes, the built-in logging mechanism of EthereumJ was co-opted to record information. Pre-

cisely, it was logged the number of pings, pong, find node and neighbour messages for the node discovery layer every minute. We also logged the number of connection attempts, authentication and handshake messages for the initial RLPx handshake every minute. In addition, for the DEVp2p layer, when its send and receive a *hello* message, the P2P version, client Id, capabilities, peer port, and peer Id was noted. The reason for any peer disconnection noted and the peer traffic volume (number of connected peers) recorded every 5 seconds. In the same manner, when the local node sent and received a *status* message, the protocol version, network Id, total difficulty, best hash, and genesis hash was logged. The number of sent and received TCP packets was recorded every ten seconds.

The results of the first deployment regarding connected peer volume, disconnection messages, discovery in/out messages, connection attempts, authentication messages, and so on are shown in chapter 6.1. A local supervisor was set to automatically start and restart the client in case of a crash. The client did not have such hang-ups. However, the local supervisor was scheduled to stop the client on the 22^{nd} of March at 10 pm.

Based on the results of the first deployment (chapter 5.1) further changes were made to the EthereumJ client. To make room for more peer connections, we increased the size of allowed peer connections from 50 to 100 and adjust the value of max open files from the default 512 to 1024. We noticed the peer processing time wasn't affected. Peer processing time could comprise of receipt retrieving, header retrieving, block retrieving, etc. EthereumJ clients normally accept incoming connections until the max peer limit is reached. A more isolated increase in the size of allowed peer connection (i.e., above 125) resulted to an increase in sent disconnections caused by *requested, user reason, ping timeout*, and with received disconnections caused by *useless peers*. These disconnection messages are as a result of processing delays by the EthereumJ node. Due to a sheer number of connected peers, the node could not adequately process and respond to all the inbound messages within the specified peer connection time limit of two seconds. As a safety measure, the size of the allowed peer connections was maintained at 100. (We could further increase the allowed peer connection above 100 without incurring peer processing delays if we increase the number of open files further, we choose not to do so due to constraint

on the hardware).

To deal with the sizeable number of too many peer messages reported earlier, another constant called *inbound_connection_too_many_peer* was defined with a value set to four minutes, and an increase in the list size to record peers that misbehave to 10000. This modification was made because we noticed the local/remote node attempting several connections with a remote/local node despite successive receptions of too many peers message milliseconds apart. We further created a *peer.blacklist* that capped peers with an incompatible protocol for a period of 30 minutes. Moreover, when a received disconnection message with reason code 0x02 (bad protocol), we record the node Id of this peer. If such messages are 20 in succession from this node, the node is added to the 'blacklist'.

An additional 25 seed nodes from discovery was added to the 25 default configuration seeds making a total number of 50 nodes to be used for discovery. The added nodes were chosen from peers that were actively in sync with our client during the first deployment. With these modifications, a second deployment of our client for a period of two months (March 23^{rd} – May 23^{rd}, 2019).

CHAPTER 6

Results and Analysis

In this Chapter, we discussed the results of our client behaviour regarding the peer traffic volume, disconnection reasons, subprotocol messages, client types, network types, nodes classification, types of services, TCP traffic volume, node geographical location and major ISP providers. In addition, we also detailed some of the modifications made to our client software.

6.1 Peer Traffic Volume

It endured approximately 12.5 hours to fully synchronized with fast sync option enabled (initial deployment at 10 pm March 18^{th}, - 10 pm March 22^{nd}, 2019). As more complex contracts have been added to the continuous chain, the later blocks are computationally expensive to properly process. The number of connected peers stayed relatively between 50 and 46 during the stable period. After the sync (at the 12.5th hour – figure 10), we had peer count between 20 and 25 for a period of 40 minutes and later 45-50 which remained constant throughout the thorough study. Note that we are not forcing the number of connected peers when the sync is over but there are lots of other peers eager to connect to us. We also noticed a sharp increase in the Ethereum subprotocols messages – figure 16, immediately after our node was fully synced. Our node was now relaying new messages after the synchronization completed.

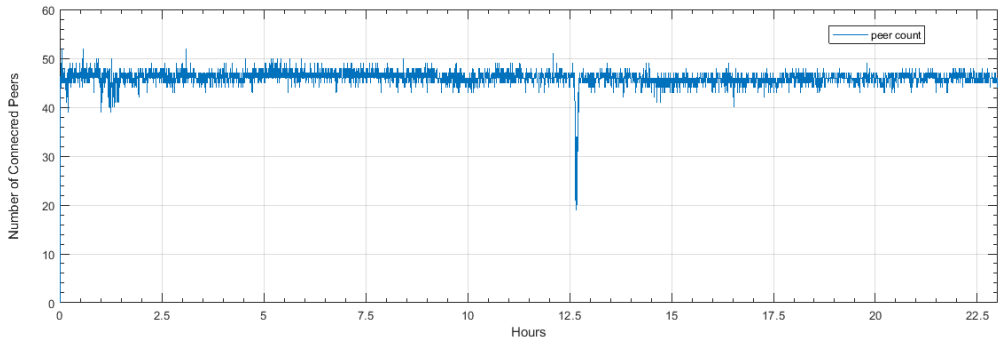

Fig10. Number of connected peers per minute. The number of connected peers stayed relatively between 46 and 50 during the stable period. We had a peer count of 20 at the 12.5hours, immediately after our node synchronized.

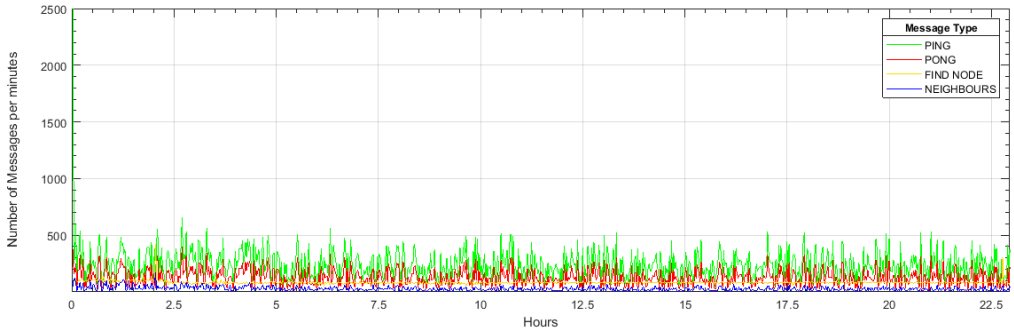

Fig11. Ping, Pong, Find node and Neighbour messages per minute.

The ping-pong pair represent the dominant message communicated for the node discovery protocol layer. During the stable period, *ping* message averaged 300-500/min while the *pong* messages received were between 100-300/min. The *find_node* message was relatively stable at 80/min while the *neighbour* messages averaged 10-80/min through the 24 hours of our first deployment – figure 11. The number of sent discoveries message (ping, pong, find_node and neighbours) was more than the received messages by approximately a factor of two – figure 12. The number of authentication messages, handshakes, and connection attempts were monitored for a period of 5.5 hours from the thirtieth minute of the client deployment. An average of 34 connection attempts, 50 handshakes and, 130 authentication messages were initiated every minute. These handshakes indicate our client is successfully exchanging messages with other peers in the network.

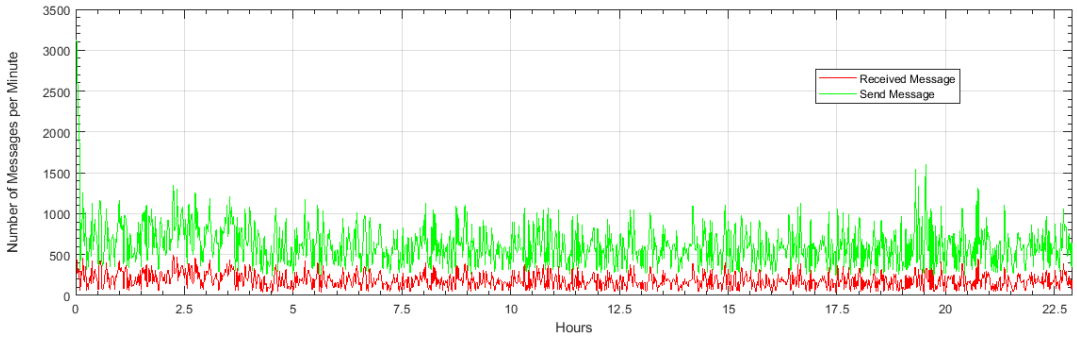

Fig12. Node discovery sent and received message. Sent discovery messages were at a factor of 2 higher than received discovery messages.

The node discovery latencies were relatively stable at 290ms – figure 13. Further analysis showed 8% of nodes at a latency of less than 45ms, 12% between 46-100ms, 21% between 100-250ms, 56% between 250-350ms and finally 3% greater than 350ms.

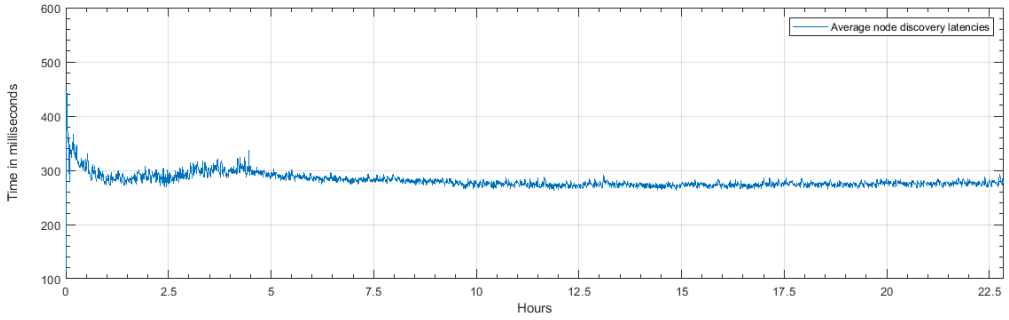

Fig13. Average node discovery in milliseconds. The node discovery time was relatively stable at 300ms.

The peer processing time alternates roughly between 1500-3000ms. The peer processing time is influenced by the state of the peer. Depending on the synchronization activity, the peer assumed the following status:

- Idle – doing nothing
- Header-retrieving: the peer retrieved the header
- Block-retrieving: the peer retrieved the block.
- Receipt-retrieving: the peer finished verifying the receipt of a specific block.

- Done-hash-retrieving: the peer has a valid and trusted copy of the block.

Peers with states of *block-retrieving* had an average processing latency greater than peer in other states

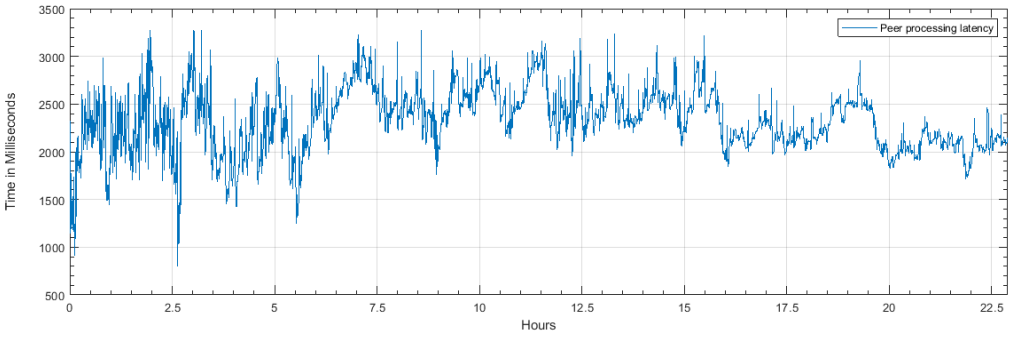

Fig14. Peer processing latencies. The value depends on the state of the peer i.e., header-retrieving, idle, etc.

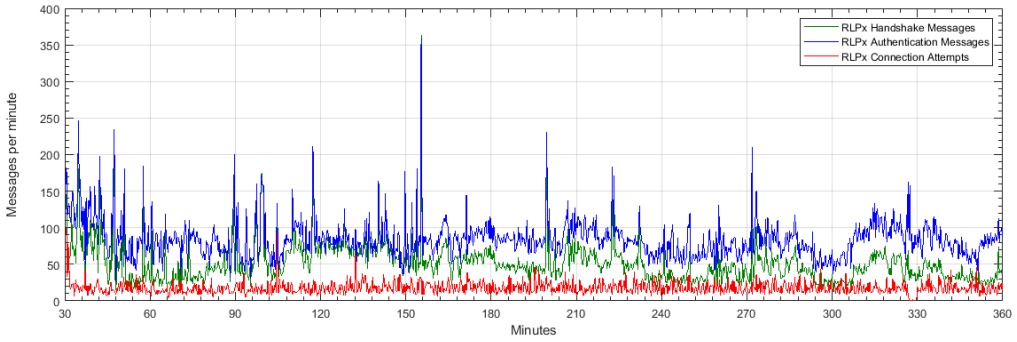

Fig15. The RLPx initiation messages. On Average we had 34 connection attempts, 75 handshakes and 130 Authentication messages per minute. The snapshot was taken at the 30th minute of deployment and for a period of 5.5 hours.

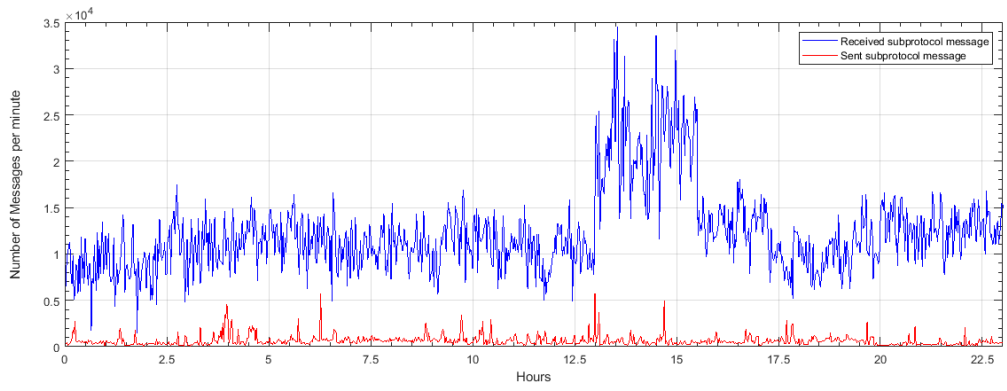

Fig16. Sent and received Ethereum subprotocol messages per minute. We observed an increased in the received messages after the synchronization that occurred at the 12.5 hours. Our client was now receiving new information.

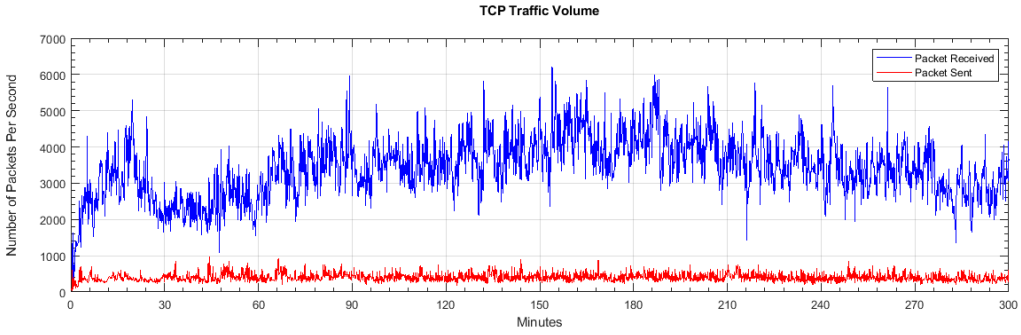

Fig17. TCP Traffic Volume measured every ten seconds: Whilst the number of packets sent per second was relatively stable at 500, packets received exhibit a greater variance with an average value of 3,000. Received packets were at a factor of six greater than sent packets.

6.2 Node Disconnections

From Table1, the dominant factor of peer disconnections for messages sent is *too many peers* followed slightly by *requested* and further by *incompatible protocol*. However, before synchronization, the dominant reason for sent peer disconnection messages was *incompatible protocol*. The most popular disconnection messages received are *user reason, too many peers* and *useless peers*. For all disconnection messages sent and received, disconnections caused by *too many peers* was at 41.5% followed by *user reason* and *requested* at 22% and 20% respectively. The reason for "too many peer" disconnections is quite understandable since we had a default peer limit of 50 and there were hundreds of other peers trying to connect to us at a given instance. To make room for more peer connections, we increased the size of allowed peer connection from 50 to 100 and change the value of max open files from the default 512 to 1024. We noticed the peer processing time wasn't affected.

After the modification made in the second deployment discussed in chapter five, disconnection messages caused by too many peers reduced and represent the second factor for sent disconnections.

REASON	SENT	RECEIVED	TOTAL	PERCENTAGE
TOO MANY PEERS	51,546	52,637	104,183	41.50%
REQUESTED	50,893	0	50,893	20.27%
UNKNOWN	215	0	215	0.087%
USER REASON	0	54,125	54,125	21.56%
USELESS PEERS	5	10,549	10,554	4.20%
INCOMPATIBLE PROTOCOL	29,652	21	29,673	11.82%
BAD PROTOCOL	0	956	956	0.38%
PING TIMEOUT	0	345	345	0.14%
PEER QUITTING	0	102	102	0.041%
-	132,311	118,735	251,046	100%

Table1. P2P Disconnection Reason for the first 50 hours. The sent disconnections message is higher than the received.

6.3 DEVp2p and Sub-protocols messages

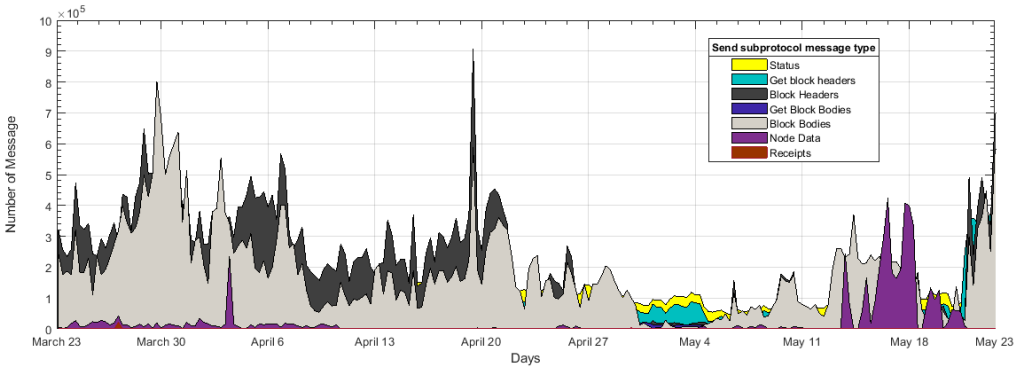

Fig18. Send Subprotocol Messages. Block headers and Block Bodies were the dominant message types.

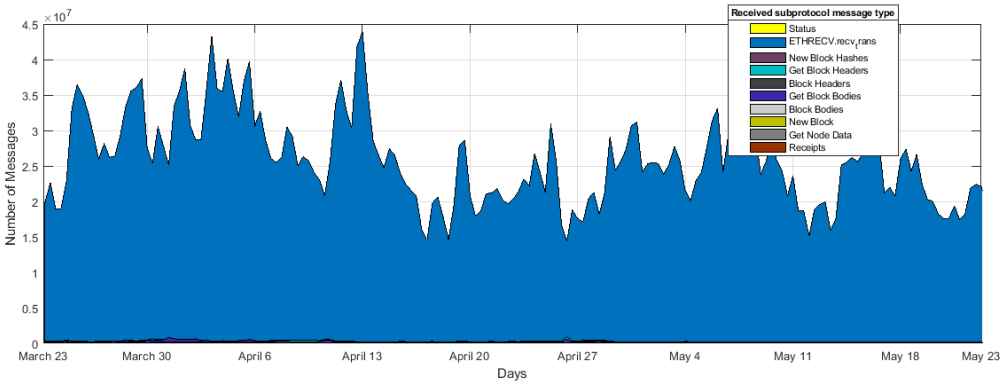

Fig19. Received Subprotocol Messages. Transaction messages was the outright message type received.

Figure 18 and *19* provide information of messages exchanged for the sub-protocol layer per day. The figures show EthereumJ client received significantly higher transactions messages than they sent. Plus, transaction message was the dominant message type of messages exchanged in the subprotocol layer. Block headers and block bodies messages dominate the sent message type in the subprotocol layers. In relation to other studies on Ethereum network [3], our client statistically sent significantly lower transactions compared to Geth and Parity clients. Also, in Kim et al. [3], transactions dominate both the sent and received messages for both Parity and Geth Clients. Geth broadcast transactions to all its peers, Parity only sent transactions to $\sqrt{n}$ peers. In EthereumJ we are sending verified transactions to all peers except the peer we received it from (according to the Yellow paper) [27]. As we could have thousands of transactions in the pending list, it's capped to some safe number (192), any peer could be able to receive from us.

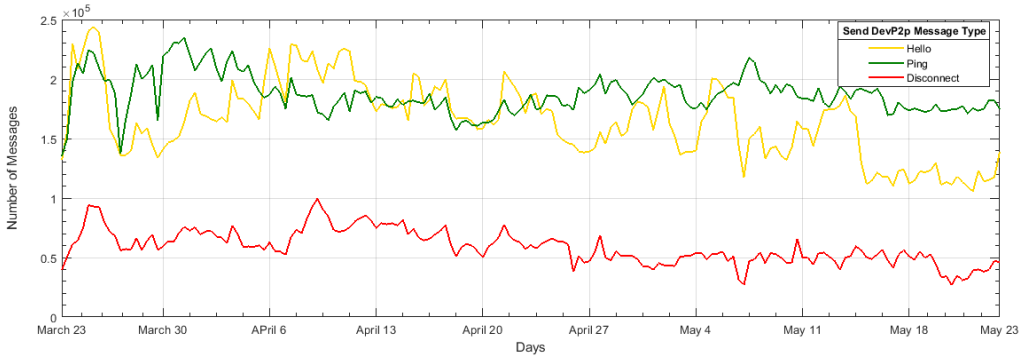

Fig20. Sent DEVp2p messages per day. Ping message type dominates messages sent.

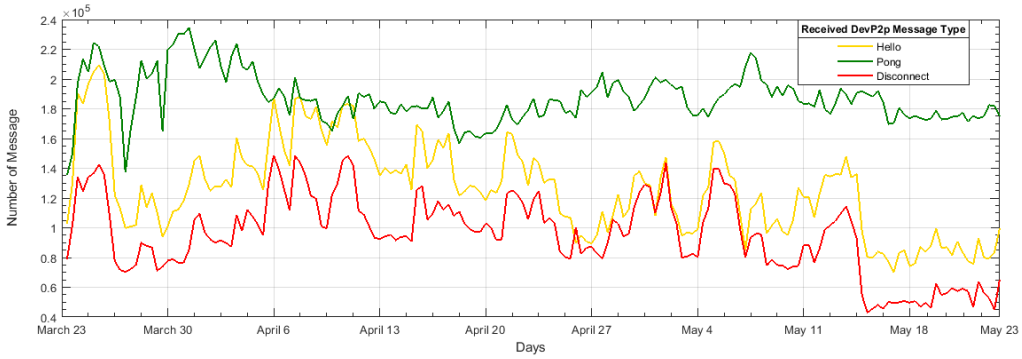

Fig 21. Received DEVp2p Messages per day. Pong message type dominates messages received.

Figure 20 and *21* highlights the message types exchanged in the DEVp2p layer. Statistically, messages sent were more than those received. The dominant message type for sent and received were the ping and pong messages. Messages sent and received by the DEVp2p wire protocol was at a factor of 10 and 100 respectively less than messages sent and received by the subprotocol services. Regarding the disconnect messages, for the sixty days of analysis, most were caused by *user reasons* (39%), *incompatible protocol* (24%), *useless peers* (18%), *too many peers* (17%) and the remaining reasons at 2%. This result is in sharp contrast to table1 were most disconnections was by too many peers.

6.4 Client Classification

For the two months of our data collection (March 23^{rd} – May 23^{rd}, 2019) our client was able to establish connections with 353,711 distinct nodes. Of those connections established, we exchanged *Hello* packets with 296,417. We identified the clients of 217,154 nodes running on more than 60 different implementations of Ethereum. Geth and Parity were the two major clients at 50% and 25% respectively while EthereumJ made up approximately 1% of nodes on the Ethereum Network. Of those clients identified, 197, 586 (>90%) run on p2pVersion5 while 19, 568 (< 10%) used p2pVersion4. Furthermore, we established synchronization with 9411 peers, with an average duration of 16 hours. The result in Table2 showed most nodes running on the Ethereum network uses either Parity or Geth client. Parity and Geth are respectively Rust and Golang official implementation of the Ethereum Network. It is a nice call by the founders of Ethereum to implement the network

NAME OF CLIENTS	NUMBER OF NODES	PERCENTAGE
GETH	110,405	50.84%
PARITY	53435	24.61%
PIRL	9419	4.34%
EGEM	3680	1.69%
GVNS	3281	1.51%
MOAC	3066	1.41%
ETHEREUMJ	3042	1.40%
GUBIQ	2419	1.11%
GMC	2376	1.09%
REOSC	1872	0.86%
GOCOIN	1511	0.70%
SWARM	960	0.44%
GSOIL	618	0.28%
PANTHEON	618	0.28%
ETHEREUMJD-DEVP2P	418	0.19%
GDBIX	397	0.18%
NETHERMIND	353	0.16%
OFBANK	296	0.14%
OTHERS (> 50 different clients)	18,988	8.74%

Table 2. Geth and Parity are the major client software at 49% and 24% respectively.

in several languages and by different persons. This has turned out to be incredibly valuable as an attack based on one client might not necessarily affect nodes running on different clients. But as evident in *Table 2*, most of the nodes on the Ethereum Network are solely based on either Geth or Parity. This indicates a vulnerability in one of these clients will greatly affect the stability of the Network. According to reports released by Security Research Labs (SRLabs), unpatched Ethereum clients pose 51% attack risk [22]. Considering *Table3*, around 34% of Geth nodes were below v.1.8.x, i.e., they need a security-critical update.

GETH VERSION	COUNT	PERCENTAGE
1.9	15,086	13.66%
1.8	54,983	49.80%
1.7	17,378	15.74%
1.6	9,318	8.44%
OTHERS	13,640	12.35%
TOTAL	110,405	100%

Table3: Geth client distribution, clients using different variant of version 1.8 were the most at 50%.

6.5 Network IDs and Genesis Hashes

We uncovered 2342 networks Ids of the 217,514 nodes with identifiable client names. Ethereum Mainnet (at 16.68%) is the dominant type on the DEVp2p Network. The top ten Ids on the DEVp2p network make up 27.5% of the total Id space, with the remaining 2332 network types making 72.50%. This reflects the diverse nature of the DEVp2p protocol. Furthermore, we could not identify the name of the network Id ***-1169308144*** and ***10088*** which are respectively the 2^nd and 6^th network Ids by popularity on the DEVp2p network.

Musicoin (7762959) is a modified version of Ethereum with more focus on Music-economy and the needs for Pay-per-play (PPP) model [37]. It supports the creation, distribution, and consumption of music in a shared economy. Ropsten, Kovan, and Rinkeby constitute the Ethereum test networks. Ropsten and Kovan implement the PoW algorithm for consensus whereas Rinkeby uses the proof of authority (PoA). In PoA, transactions are verified by approved accounts known as validators. Validators run the software (automated) which allows them to sequence transactions into blocks. Ropsten and Kovan being an open PoW are more decentralized than Rinkeby. However, given the lower hashing being used to mine the blocks, it wouldn't take much to ruin. All the test nets shared the commonality of no financial benefits. Ethergem (Egem) is a community supported coin with no pre-mine. This coin is based off of Ethereum and without the limitations like limited transactional throughput. It provides query nodes that pay dividends to node holders [39]. To-

Network ID	Network Name	Genesis Hash	Number	Percentage
1	Homestead	d4e56740.....b1cb8fa3	36,218	16.68%
-1169308144	Unknown	29a742ba.....9aa1a273	3635	1.67%
1313114	Ether-1	70cefc67.....0c32e653	3223	1.48%
3	Ropsten	d4e56740.....b1cb8fa3	3189	1.47%
42	Kovan	426ab4e1.....4ec0a0c9	2699	1.24%
10088	Unknown	4d680489.....b223c56b	2567	1.18%
7762959	Musicoin	4eba28a4.....4ac952e9	2170	1.00%
1987	Egem	dfc3e94e.....fd9a4149	2072	0.95%
88	Tomochain	406f1b7.....047418af	2041	0.94%
4	Rinkeby	6341fd3d.....5767e177	1927	0.89%
-	-	2332 different network Ids	157,413	72.50%
-	-	-	TOTAL (217,154)	100%

Table 4. Network types on DEVp2p Network.

mochain is a public EVM compactible blockchain with a low transaction fee, fast confirmation time, double validation and randomization for security guarantees [40]. It uses proof of stake voting (PoSV) consensus, which is PoS-based protocol with a fair voting mechanism, rigorous security guarantees, and fast finality.

6.6 Nodes Classification

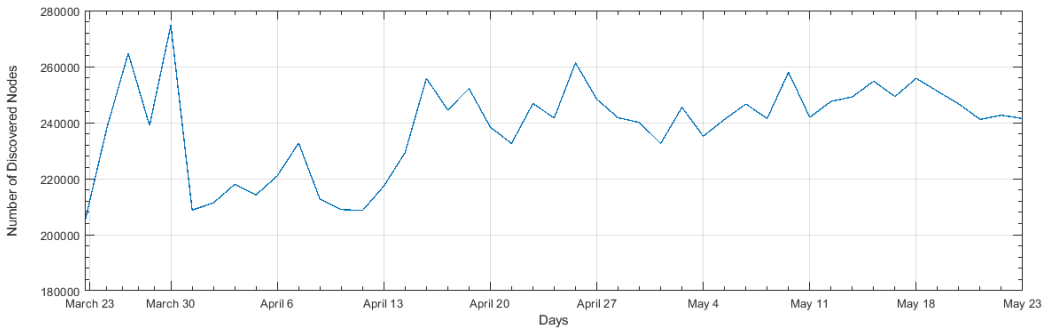

Fig22. Number of nodes discovered per day.

On average, our client made 240,223 node discoveries per day. A maximum node count was registered on March 30^{th} at 275, 011 whilst a minimum count of 205,371

was recorded on March 23^{rd} – the first day of our analysis. We also logged the number of discovered nodes belonging to the Mainnet network every minute. A data structure was defined to contain the logged information. This data structure has a capacity of 5,000 nodes, and it periodically refreshes every second. This number of nodes is far away from the combined number of nodes of the network, and thus the computation can only approximate the values the whole network would exhibit. The classification of nodes in figure 23 is based on the description in *chapter 4.2*. In contrast to the chapter, we merged the 'discovered' nodes with the 'alive' ones. Furthermore, we considered the 'evicted-candidates' as dead nodes. While most of the nodes were alive, our scan revealed an average number of 100 active nodes per minute throughout the continuous study.

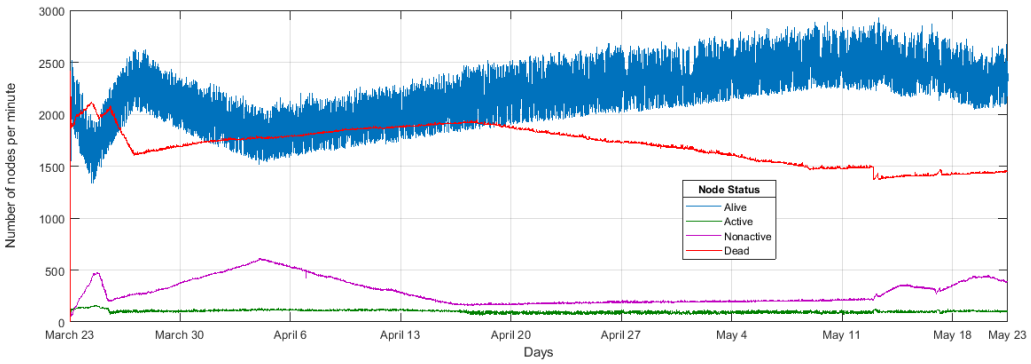

Fig23. Node status of nodes belonging to the Mainnet network. Discovered nodes are group with alive nodes whilst evicted nodes are classified as dead.

6.6.1 Classification of nodes by service

We identified the following subprotocol services on the network: *eth, etz, shh, bzz, mc, les, vns, dex, istanbul, hive, stream, pss, mit, exp, smilobft, par, ur, ddm, glx, pip, pchain, adh, and ele, lax, wsh, pwh, ctxc, anr, okc, wolk, vch.*

Pip is a parity light protocol. PIP client allows a verifiable on-demand data fetching efficiently, and it is compatible with clients that store almost nothing and those that store everything [66]. Its goal is to limit any given request to one or two round-trip-time in the network. Par [67] is an extension of the eth subprotocol. It involves sending snapshots over the network to get the full state of a block more quickly, and then filling in the blocks from the genesis to the snapshot in the background.

Services type	Number of Nodes	Percentage
Eth	191,825	88.19%
Swarm	7129	3.28%
Etherzero (etz)	1473	0.68%
Whisper (shh)	1309	0.60%
Istanbul	1302	0.60%
Others	14476	6.66%
Total	217,514	100%

Table 5. Classification of nodes by service.

We categorized *pip, par, les* as subservices of *eth*. Whisper (ssh) is a distributed messaging service. Swarm (bzz, pss, hive) is a decentralized storage service. *Istanbul, Expanse (exp), Etherzero(etz), mc* are Ethereum-like services known as forks. As of this writing, two updates have been approved for the *Istanbul* hard fork scheduled for October of 2019 [69]. These updates are EIP 2024 and EIP 1702. EIP introduces a new precompile for a new hash function known as Blake2. The function is said to be faster in verifying and authenticating blockchain than SHA-3. EIP 1702, authored by Parity Technologies developer Wei Tang is geared towards upgradability. Wei Tang suggests a methodology for hard fork known as "account versioning." He explained in his proposal: "By allowing account versioning, we can execute different virtual machines for contracts created at different times. This allows breaking the features to be implemented while making sure existing contracts work as expected."

6.6.2 Nodes geographical distribution and ISP providers

The ExtremeIp API (*https://extreme-ip-lookup.com/*) was used to register the geographical distribution of nodes and get the ISP providers of addresses residing on the Ethereum Mainnet Network. We gained connections with nodes across 124 countries on 2,370 ISP providers. The United States of America, at 26.74% represent the location of most nodes on the network followed by China and Germany at 18.93% and 6.89% respectively – Table 6. The top three countries made up more than half (53%) of all Mainnet nodes while the bottom 109 countries were at 18%. In terms of ISP providers, 20% of all nodes were serviced by Amazon at first followed by ChinaNet

and Alibaba at 9% and 6% respectively. Note that in the classification provided on *Table7*, we did not distinguish an ISP provider and its subsidiaries. For example, in the categorization of Alibaba ISP provider, we categorized its subsidiaries like Alicloud-hk, Aliyun as Alibaba service providers. Interesting, approximately 90% of disconnections caused by ping timeout, bad protocol and peer quitting were from nodes not residing in the top 10 ISP providers listed on *Table7*. We concluded further analysis of the Mainnet nodes and uncovered a whopping 63.41% of nodes resided on data centers. Intriguing, more than 90% of all nodes registered from Singapore, Hong Kong and South Korea were on data centers. The structure of the P2P network impacts security and performance for digital assets like Ethereum [4]. A geographically clustered network can quickly propagate a new block to many other nodes. This makes it more difficult for a malicious miner to propagate conflicting blocks to many other nodes quicker than altruistic nodes. Moreover, a less clustered network may mean full nodes are being run by a vaster variety of users – good for decentralization.

Name of Country	Number of Nodes	Percentage
United States of America	9685	26.74%
China	6856	18.93%
Germany	2497	6.89%
Japan 1858	5.13%	
South Korea	1220	3.37%
Russia	1159	3.32%
Singapore	1122	3.10%
Netherlands	950	2.62%
France	913	2.52%
United Kingdom	807	2.23%
Canada	650	1.79%
Ireland	554	1.53%
Ukraine	453	1.25%
Hong Kong	424	1.17%
India	402	1.11%
109 other countries	6664	18.40%
-	TOTAL (36,218)	100%

Table 6. Geographical distribution of nodes on the Mainnet network.

ISP Providers	Number of IPs	Percentage
Amazon	7075	19.53%
ChinaNet	3362	9.28%
Alibaba	2341	6.46%
Google	1659	4.58%
NTT	1394	3.85%
Hetzner	1071	2.96%
OVH	1069	2.95%
Choopa	905	2.50%
Microsoft	812	2.24%
China Unicom	750	2.07%
2360 More ISP providers 14875	41.07%	
-	TOTAL (36218)	100%

Table 7. Distribution of Mainnet (Homestead) IP addresses by ISP providers.

6.7 Block Propagation

We listened to various nodes on the Ethereum Network and registered the blocks being broadcast and the timestamp of when we received the block. We monitored the block propagation for ten days (240 hours – 864,000 seconds). During this period, we received a grand total of 1,714,583 blocks from 1325 unique peers. From the Ethereum Yellow Paper [27], the adjusted time for a new block is within 12-15 seconds. With this knowledge and taking the upper bound of 15 seconds, we expect around 57,600 new blocks for a period of ten days. The 1,714,583 blocks received is an exaggerated estimate of the number of blocks mined during this period. The reason for this huge difference is we received duplicate copies of blocks and sometimes old copies which were not in the interval of our measurement. For the specific instances, blocks with number *3532412, 4523223, 6341223*, were registered while our chain was at the 7898788^th block. To this effect, we nothing but considered blocks which were within the interval of measurement. We later filtered blocks with duplicated copies. During the filtering, a duplicated block with a higher difficulty level was selected over a block with lesser difficulty. Furthermore, when two blocks

had the same difficulty level, the one first received was considered. After the filtering, a total of 69,190 blocks remained. Using this number, a unique block is received every 12.49 second. This result aids us with the synchronization state of our chain. In a separate analysis, we considered duplicated blocks within the interval of our analysis. We, however, did not consider blocks which were within the range of measurement but received 30 seconds after the inscription of the first block copy. With this consideration, 1,410,172 blocks remained for the 10 days period. When we follow the assumption that nodes will not send the same block copy to the same client. Hence, within 12.49 seconds, we have received a new block from approximately 20 clients. Our network statistics showed we maintained an average of 100 active peers during our measurement. With this information, we can hence say an approximated 20% of connected peers possessed an updated copy of the blockchain at any given instance during our measurement.

CHAPTER 7

Conclusion and Future Work

In this thesis, we have analyzed the peer-to-peer (P2P) network of Ethereum over a period of two months to gain an insight into this previously opaque network. By deploying an Ethereum client and making use of the RLPx and DEVp2p protocols, we find and carefully collect node information. We utilized these information to understand precisely the underlying protocol of this network. Our client established peer connections with 9,411 nodes with an average connection time of 16 hours. An approximate 100 active connections were observed at any instant throughout the thorough study. Of the 100 active peers maintained, 20% were at the latest block of the chain. A new block was received at an average of 12.48 seconds. Most nodes disconnections were caused by user reason — 39%, incompatible protocol — 24%, useless peer — 18% and too many peers — 17%. Results further showed Ethereum services made 88% of all services residing in the DEVp2p protocol layer. A further 2342 network Ids were uncovered in the DEVp2p protocol layer with Ethereum Mainnet the most at 16%. For the node geographical distribution, we established connections across 124 countries and 2,370 ISP providers on the Mainnnet Network. Although this appears to represent a balanced distribution, we have observed the top 3 countries occupying more than 50% of all nodes on the Mainnet Network. Further, more than 63% of all Mainnet nodes were residing in data centers.

This brief analysis of the information presented in this report allows us to make some guidelines for prospective studies. One instance could be a criterion for a

miner's inclusion of a specific transaction in a block, and the time until a transaction is included into a block. A more in-depth information propagation analysis can be performed by increasing the amount of data collected and the number of connections made to listen to the network. Another area for future work is enabling automatic update for Ethereum clients. Ethereum clients would benefit from automatic updates since vulnerability which mostly results in financial loss can be appended as soon as possible.

References

[1] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[2] Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., ... & Rosu, G. (2018, July). Kevm: A complete formal semantics of the ethereum virtual machine. In 2018 IEEE 31st Computer Security Foundations Symposium (CSF)(pp. 204-217). IEEE.

[3] Kim, S. K., Ma, Z., Murali, S., Mason, J., Miller, A., & Bailey, M. (2018, October). Measuring Ethereum Network Peers. In Proceedings of the Internet Measurement Conference 2018(pp. 91-104). ACM.

[4] Gencer, A. E., Basu, S., Eyal, I., van Renesse, R., & Sirer, E. G. (2018). Decentralization in bitcoin and ethereum networks.arXiv preprint arXiv:1801.03998.

[5] Ethereum Foundation (February 2019)- https://www.ethereum.org/

[6] Vujičić, D., Jagodić, D., & Ranđić, S. (2018, March). Blockchain technology, bitcoin, and Ethereum: A brief overview. In NFOTEH-JAHORINA (INFOTEH), 2018 17th International Symposium(pp. 1-6).IEEE.

[7] Dhillon, V., Metcalf, D., & Hooper, M. (2017). Unpacking Ethereum. In Blockchain Enabled Applications (pp. 25-45). Apress, Berkeley, CA.

[8] Blockchain4Innovation - https://www.blockchain4innovation.it/esperti/cose-quali-gli-ambiti-applicativi-ethereum/

[9] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. white paper.

[10] Dwork, C., Goldberg, A., & Naor, M. (2003, August). On memory-bound functions for fighting spam. In the Annual International Cryptology Conference (pp. 426-444). Springer, Berlin, Heidelberg.

[11] A guide to 99% fault consensus - https://vitalik.ca/general/2018/08/07/99_fault_tolerant.ht

[12] CoinMarketCap. 2019. Cryptocurrency Market Capitalizations. Retrieved September 18, 2019 from https://coinmarketcap.com/

[13] Node Discovery Protocol ethereum – wiki https://github.com/ethereum/wiki/wiki/Node-discovery-protocol

[14] Node Discovery Protocol v4 - https://github.com/ethereum/devp2p/blob/master/discv4.md

[15] DEVp2p Application protocol https://github.com/ethereum/devp2p/blob/master/devp2p.m

[16] Martínez, V. G., & Encinas, L. H. (2010, August). A Comparison of the Standardized Versions of ECIES. In Information Assurance and Security (IAS), 2010 Sixth International Conference on (pp. 1-4). IEEE.

[17] Light Ethereum sub-protocol- https://github.com/paritytech/wiki/blob/master/Light-Ethereum sub-protocol-(LES).md

[18] Ethereum-go swarm https://github.com/ethereum/go-ethereum/tree/master/swarm

[19] Ethereum wiki- Whisper - https://github.com/ethereum/wiki/wiki/Whisper

[20] Katkuri, S. (2018). A survey of data transfer and storage techniques in prevalent cryptocurrencies and suggested improvements.arXiv preprint arXiv:1808.03380.

[21] The RLPx Transport Protocol-https://github.com/ethereum/devp2p/blob/master/rlpx.md# discovery

[22] Security Research labs: The blockchain ecosystem has a patch problem https://srlabs.de/bites/blockchain_patch_gap/

[23] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., & Danezis, G. (2017). Chainspace: A sharded smart contracts platform. arXiv preprint arXiv:1708.03778.

[24] Kevin Leffew, February 2019 - A Brief Overview of Kademlia, and its use in various decentralized platforms

[25] Petar Maymounkov and David Mazières (2002) - Kademlia: A peer-to-peer information system based on XOR metric

[26] Péter Szilágyi - Eth/63 fast synchronization algorithm - https://github.com/ethereum/go-ethereum/pull/1889

[27] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, 151, 1-32.

[28] Li, X., Jiang, P., Chen, T., Luo, X., & Wen, Q. (2017). A survey on the security of blockchain systems. Future Generation Computer Systems.Baliga, A. (2017). Understanding blockchain consensus models. In Persistent.

[29] Azouvi, S., McCorry, P., & Meiklejohn, S. (2018). Betting on blockchain consensus with fantomette. arXiv preprint arXiv:1805.06786.

[30] Abraham, I., & Malkhi, D. (2017). The blockchain consensus layer and BFT. Bulletin of EATCS, 3(123).

[31] Cachin, C., & Vukolić, M. (2017). Blockchain consensus protocols in the wild. arXiv preprint arXiv:1707.01873.

[32] Wüst, K., & Gervais, A. (2016). Ethereum eclipse attacks. ETH Zurich. [106]

[33] Ekparinya, P., Gramoli, V., & Jourjon, G. (2018, October). Impact of man-in-the-middle attacks on ethereum. In 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS) (pp. 11-20). IEEE.

[34] Tosh, D. K., Shetty, S., Liang, X., Kamhoua, C., & Njilla, L. (2017, October). Consensus protocols for blockchain-based data provenance: Challenges and opportunities. In 2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON) (pp. 469-474). IEEE. [108]

[35] Buterin, V., & Griffith, V. (2017). Casper the friendly finality gadget. arXiv preprint arXiv:1710.09437.

[36] Mayer, H. (2016). ECDSA security in bitcoin and ethereum: a research survey. CoinFaabrik, June 28, 126.

[37] Kendler, E. A., Zohar, A., & Goldberg, S. (2015). Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In 24th USENIX Security Symposium (USENIX Security 15).

[38] https://github.com/Musicoin/go-musicoin/wiki/The-changes-of-Musicoin-Blockchain-from-Ethereum

[39] Ethergem: A community based blockchain -2019 https://egem.io/

[40] Team, T. R. D. (2018). TomoChain: Masternodes Design Technical White Paper Version 1.0.

[41] Milutinovic, M., He, W., Wu, H., & Kanwal, M. (2016, December). Proof of luck: an efficient blockchain consensus protocol. In proceedings of the 1st Workshop on System Software for Trusted Execution (p. 2). ACM.

[42] Mattila, J. (2016). The blockchain phenomenon–the disruptive potential of distributed consensus architectures (No. 38). The Research Institute of the Finnish Economy.

[43] Kraft, D. (2016). Difficulty control for blockchain-based consensus systems. Peer-to-Peer Networking and Applications, 9(2), 397-413.

[44] Zheng, Z., Xie, S., Dai, H., Chen, X., & Wang, H. (2017, June). An overview of blockchain technology: Architecture, consensus, and future trends. In 2017 IEEE International Congress on Big Data (BigData Congress) (pp. 557-564). IEEE.

[45] Garay, J., Kiayias, A., & Leonardos, N. (2015, April). The bitcoin backbone protocol: Analysis and applications. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 281-310). Springer, Berlin, Heidelberg.

[46] Clement, A., Wong, E. L., Alvisi, L., Dahlin, M., & Marchetti, M. (2009, April). Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In NSDI (Vol. 9, pp. 153-168).

[47] Watanabe, H., Fujimura, S., Nakadaira, A., Miyazaki, Y., Akutsu, A., & Kishigami, J. J. (2015, October). Blockchain contract: A complete consensus using blockchain. In 2015 IEEE 4th global conference on consumer electronics (GCCE) (pp. 577-578). IEEE.

[48] Atzei, N., Bartoletti, M., & Cimoli, T. (2017, April). A survey of attacks on ethereum smart contracts (sok). In International Conference on Principles of Security and Trust (pp. 164-186). Springer, Berlin, Heidelberg.

[49] Wüst, K., & Gervais, A. (2016). Ethereum eclipse attacks. ETH Zurich.

[50] Chaum, D. L. (1981). Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM, 24(2), 84-90.

[51] A. Miller and J. J. LaViola Jr. Anonymous Byzantine consensus from moderately-hard puzzles: A model for Bitcoin. 2014.

[52] Sankar, L. S., Sindhu, M., & Sethumadhavan, M. (2017, January). Survey of consensus protocols on blockchain applications. In 2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS) (pp. 1-5). IEEE.

[53] Schwartz, D., Youngs, N., & Britto, A. (2014). The ripple protocol consensus algorithm. Ripple Labs Inc White Paper, 5.

[54] Koshy, P., Koshy, D., & McDaniel, P. (2014, March). An analysis of anonymity in bitcoin using p2p network traffic. In International Conference on Financial Cryptography and Data Security (pp. 469-485). Springer, Berlin, Heidelberg.

[55] Decker, C., & Wattenhofer, R. (2013, September). Information propagation in the bitcoin network. In IEEE P2P 2013 Proceedings (pp. 1-10). IEEE.

[56] Donet, J. A. D., Pérez-Sola, C., & Herrera-Joancomartí, J. (2014, March). The bitcoin P2P network. In International Conference on Financial Cryptography and Data Security (pp. 87-102). Springer, Berlin, Heidelberg.

[57] Velde, F. (2013). Bitcoin: A primer.

[58] Baumann, A., Fabian, B., & Lischke, M. (2014, April). Exploring the Bitcoin Network. In WEBIST (1) (pp. 369-374).

[59] Luu, L., Chu, D. H., Olickel, H., Saxena, P., & Hobor, A. (2016, October). Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (pp. 254-269). ACM.

[60] Bartoletti, M., & Pompianu, L. (2017, April). An empirical analysis of smart contracts: platforms, applications, and design patterns. In International Conference on Financial Cryptography and Data Security (pp. 494-509). Springer, Cham.

[61] Alharby, M., & van Moorsel, A. (2017). Blockchain-based smart contracts: A systematic mapping study. arXiv preprint arXiv:1710.06372.

[62] Wohrer, M., & Zdun, U. (2018, March). Smart contracts: security patterns in the ethereum ecosystem and solidity. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 2-8). IEEE.

[63] Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2016, May). Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In 2016 IEEE symposium on security and privacy (SP) (pp. 839-858). IEEE.

[64] Merkle, R. C. (1989, August). A certified digital signature. In Conference on the Theory and Application of Cryptology (pp. 218-238). Springer, New York, NY.

[65] Pilkington, M. (2016). 11 Blockchain technology: principles and applications. Research handbook on digital transformations, 225.

[66] The Parity Light Protocol - https://wiki.parity.io/The-Parity-Light-Protocol-(PIP)

[67] Warp Sync (par) - https://wiki.parity.io/Warp-Sync

[68] https://hackernoon.com/the-ethereum-blockchain-size-has-exceeded-1tb-and-yes-its-an-issue-2b650b5f4f62

[69] Christine Kim, June 2019 https://www.coindesk.com/ethereum-devs-approve-first-2-code-changes-for-istanbul-hard-fork.

[70] Ethhub: Proof of Stake- https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/

[71] https://github.com/ethereumbook/ethereumbook/blob/develop/03clients.asciidoc#requirements

[72] https://medium.com/mit-media-lab-digital-currency-initiative/medrec-electronic-medical-records-on-the-blockchain-c2d7e1bc7d09

[73] https://ens.domains/

[74] Mizrahi, A. (2015). A blockchain-based property ownership recording system. A Blockchain-based Property Ownership Recording System.

[75] Francisco, K., & Swanson, D. (2018). The supply chain has no clothes: Technology adoption of blockchain for supply chain transparency. Logistics, 2(1), 2.

[76] Hanifatunnisa, R., & Rahardjo, B. (2017, October). Blockchain based e-voting recording system design. In 2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA) (pp. 1-6). IEEE.

[77] Kaaniche, N., & Laurent, M. (2017, October). A blockchain-based data usage auditing architecture with enhanced privacy and availability. In 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA) (pp. 1-5). IEEE.

[78] Kiffer, L., Levin, D., & Mislove, A. (2017, November). Stick a fork in it: Analyzing the Ethereum network partition. In Proceedings of the 16th ACM Workshop on Hot Topics in Networks (pp. 94-100). ACM.

[79] Dannen, C. (2017). Introducing Ethereum and Solidity (p. 185). Berkeley: Apress.

[80] Bahga, A., & Madisetti, V. K. (2016). Blockchain platform for industrial internet of things. Journal of Software Engineering and Applications, 9(10), 533.

[81] Feld, S., Schönfeld, M., & Werner, M. (2014). Analyzing the Deployment of Bitcoin's P2P Network under an AS-level Perspective. Procedia Computer Science, 32, 1121-1126.

[82] Marcus, Y., Heilman, E., & Goldberg, S. (2018). Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network. IACR Cryptology ePrint Archive, 2018, 236.

[83] Ethereum Improvement Proposal - https://eips.ethereum.org/

[84] Moreno-Sanchez, P., Modi, N., Songhela, R., Kate, A., & Fahmy, S. (2018, April). Mind your credit: Assessing the health of the ripple credit network. In Proceedings of the 2018 World Wide Web Conference (pp. 329-338). International World Wide Web Conferences Steering Committee.

[85] Henningsen, S., Teunis, D., Florian, M., & Scheuermann, B. (2019, June). Eclipsing Ethereum Peers with False Friends. In 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (pp. 300-309). IEEE.

[86] Lago Sestrem O., Rafael A. Piemontez, Lucas A Martins, Valderi Leithardt, Cesar A. Zeferino. (2019, May). Experimental Analysis of the Processing Cost o the Ethereum Blockchain in a Private Network.

[87] Nadi Sarrar, (2019). On transaction parallelizability in Ethereum

[88] Topcuoglu, H., Hariri, S., & Wu, M. Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE transactions on parallel and distributed systems, 13(3), 260-274.

[89] Ferretti, S., & D'Angelo, G. (2019). On the Ethereum blockchain structure: A complex networks theory perspective. Concurrency and Computation: Practice and Experience, e5493.