Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

The Peer Discovery Layer of the Ethereum Network

Semester Thesis

Jérôme Landtwing

jeromela@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Dr. Lucianna Kiffer
Prof. Dr. Roger Wattenhofer

January 12, 2024

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Lucianna Kiffer for the supervision and guidance throughout this project. During our weekly meetings her expertise helped me to develop a deepened understanding of the subject and to shape the output of this research work. I am heartily thankful for her support and patience throughout the journey of this project.

Abstract

The blockchain domain advanced significantly in recent years. While already in broad use, there is still potential for technical improvements. Ethereum introduced *The Merge* in late 2022 and transitioned away from traditional Proof-of-Work to the more energy efficient Proof-of-Stake. From then onward Ethereum consisted of two distinct networks: the execution layer and the consensus layer. The consensus layer has been explored in previous work.

We focus our research on the peer-discovery of Ethereum. Our contribution is the development of a crawler which was used to gather fine-meshed data on the participants of the execution layer as well as their routing tables. As a summary, this project highlights the structures of the Ethereum network and its discovery protocol. The most recent crawl discovered a total of 236'606 unique enodes, of which 42'363 were responsive. 86% of the active enodes stay in the network for more than a week. On average an active node holds 147 unique enodes in its local table.

Contents

Chapter 1

Introduction

1.1 Motivation

The motivation for this thesis evolved from the new dynamics created by Ethereums *Merge*. Following *The Merge* the Ethereum protocol consists of two distinct networks: The Beacon Chain responsible for the consensus and the execution layer responsible for transactions and state management. In this work we aim to explore and understand the execution network of Ethereum, connecting our results to the findings of the recently explored Beacon Chain [1].

The key questions we aim to answer include metrics on how many nodes participate in the execution network, how many of them are responsive and how long active nodes stay in the network. For this purpose we aim to build a crawler which explores the execution layer of Ethereum. Delving into the discovery protocol to get a complete view of a node's local table. Assembling this knowledge into a network crawler traversing the local table of all nodes.

Chapter 2

Background

2.1 The Merge

Originally the Ethereum mainnet was secured by proof-of-work. The Beacon Chain first existed as a separate blockchain introducing proof-of-stake to Ethereum. In September 2022 *The Merge* was the event where the Beacon Chain was integrated into the original execution layer of Ethereum. Thus transitioning from proof-of-work to proof-of-stake with a promised reduction of the consumed energy by 99.9%. Following *The Merge* the Ethereum network consists of two different layers (but not two different chains anymore): the consensus layer and the execution layer. The consensus layer is used to verify the consensus while all the transactions happen in the execution layer. The execution layer's network is divided into two stacks: The discovery network stack is built on top of UDP and is used to discover and find peers. Transactions and blocks are exchanged over the peer-to-peer network in the DevP2P stack which runs over TCP.

2.2 Enodes

Every node in the Ethereum network has its own (32 byte) private and public key pair on the secp256k1 elliptic curve. The 64 byte node-ID is the Keccak-256 hash of the public key and is used as the primary identifier. The format used to describe Ethereum nodes in the execution layer is called *Enode*. Enode is an URL address format used to identify Ethereum nodes [2]. The present identifiers are:

- **node-ID**: hexadecimal representation of the node-ID derived from the public key.
- **hostname**: the node's IP address. While it is possible to use IPv6 addresses, the vast majority uses the IPv4 address.
- **port**: describes the TCP listening port.

- **discport** (optional): If the TCP and UDP port (used for discovery) don't have the same value, the discovery port is passed with the additional discport keyword.

These values are assembled in the format:

```
enode://node-ID@IP:TCP-port?discport=UDP-port
```

The following example, describes the Enode of a node with (fictive) node-ID deadc0ffee, IP address 10.3.58.6, TCP port 30303 and UDP discovery port 30301.

```
enode://deadc0ffee@10.3.58.6:30303?discport=30301
```

2.3 Distributed Hash Tables

Kademlia [3] is an application of Distributed Hash Tables (DHT). In Kademlia every node organizes it's local table in buckets classified by the XOR distance metric:

$$d(n_1, n_2) = n_1 \oplus n_2$$

The i-th bucket of node n's routing table is the collection of all neighbors n_x with distance $2^i \leq d(n, n_x) < 2^{(i+1)}$. Depending on the client specification k-entries are kept per bucket. When a new node is encountered, it is inserted into the corresponding bucket. In case this bucket is full the specification [4] suggests to remove the least recently seen node if it does not respond to a ping. If node_i wants to locate node_j in the network, it sends a location request for node_j to one of its peers. This peer looks up in which bucket node_j would fall into and returns entries from this bucket. Then node_i can relay the request to one of the nodes in the response getting closer to node_j with each iteration. This ensures that in a network with n nodes every node can locate any other node by within $\mathcal{O}(\log(n))$ steps.

2.4 Node Discovery Protocol (Discv4)

The underlying principle of the discv4 protocol makes use of a modified form of Kademlia. The goal of the protocol is to enable the discovery of other participants in the peer-to-peer network. According to statistics of *ethernodes.org* – "The Ethereum Network & Node explorer" – *go-etherum* is the most popular client implementation [5]. Therefore this thesis focuses on *go-ethereum*'s implementation of the discovery protocol. The code is open source and available on github [6]. All clients follow the Ethereum protocol which ensures that they can communicate with each other. This allows the discovery of nodes regardless which client they are running, while the guarantee that we query their full table

(as described in sec 3.1) does not necessarily hold for nodes running clients other than go-etherum.

The most important packet-types are the Ping, Pong, FindNode and Neighbors packets. The version field describes the IP version while it is possible to use IPv6 this option is used only by a handful of nodes. The vast majority of traffic in the Ethereum network uses IPv4. The expiration field is a UNIX time stamp, if the time stamp lies in the past the packet is discarded.

```
Ping Packet:
packet-data = [version, from, to, expiration, ...]
from = [sender-ip, sender-udp-port, sender-tcp-port]
to = [recipient-ip, recipient-udp-port, 0]

Pong Packet:
packet-data = [to, ping-hash, expiration, ...]
```

When a Ping packet is received, the recipient should reply with a Pong packet. Further the node can be added to the recipient's local table. Pong is the reply to a Ping packet, the ping-hash is the hash of the Ping packet. A Pong should be ignored if the hashes mismatch. The exchange of Ping and Pong is executed over UDP. The successful exchange of Ping and Pong is called bonding. A FindNode packet requests information about nodes close to target (identified by its public key). The response to a FindNode packet contains the 16 closest nodes to the target from its local table assembled in a Neighbors packet.

```
FindNode Packet:
packet-data = [target, expiration, ...]

Neighbors Packet:
packet-data = [nodes, expiration, ...]
nodes = [[ip, udp-port, tcp-port, node-id], ...]
```

Clients come with a handful of hardcoded bootnodes whose purpose is to serve as an entry point for new nodes connecting to the network. After bonding to the bootnode the node can send a FindNode request to it (using a random public key or it's own public key for example). The response contains a list of peers which the new node can (try) to connect to. The same procedure can be repeated with the newly added peers.

Chapter 3

Gathering Network Data

To better understand the execution layer of Ethereum our goal was to collect a dataset of network participants and their routing tables over time. This involved the development of a crawler capable of querying node's full routing tables. The network changes at a high frequency: nodes change their node-ID or their hostname, nodes go offline and new nodes join the network. To get a meaningful snapshot, we set the limit for crawling the full network to 30 minutes. To dig further into the network's properties the crawler was extended with a connectivity checker tracking the responsiveness of discovered participants. The connectivity checker collects connectivity metrics based on responsiveness over the Ethereum network and the ICMP ping. The crawler was scheduled to run daily, the connectivity checker was scheduled to run on a hourly basis.

3.1 Quering node's local tables

The first task for the crawler is querying all buckets of a node's local table. To achieve this, it sends at least one Findnode query with a target that falls into each bucket.

To query a bucket, the crawler needs a public key, that will fall into this bucket, figure 3.1 shows the relation between node-IDs and buckets. While the relation between public keys and node-IDs is determined by the hash function.

	XOR mask	node-ID to query
bucket 0	0x 0000'0000	node-ID $\oplus$ 0x00
bucket 1	0x 1000'0000	node-ID $\oplus$ 0x80
bucket 2	0x 0100'0000	node-ID $\oplus$ 0x40
bucket 3	0x 0010'0000	node-ID $\oplus$ 0x20
bucket 4	0x 0001'0000	node-ID $\oplus$ 0x10
bucket 5	0x 0000'1000	node-ID $\oplus$ 0x08
...	...	...

Figure 3.1: Relation between buckets and bit-flips in the node-ID.

The Neighbors packet (visualized in section 2.4) is defined to carry 16 neighbors, in practice it is common that nodes store more than 16 nodes in each bucket of their local table. To query the node's full table the crawler queries each bucket multiple times with different targets as input until no further new neighbors are returned. Heuristically we determined that after three queries no new nodes were returned from the corresponding bucket. To save time during the crawl, the relation of node-IDs and public keys was pre-computed and stored in a lookup table. Limited by computational resources the lookup table comprises $3*2^{14}$ public keys which cover the 14-bit prefixes from 0x0000 to 0xfffc in the node-IDs, for each possible prefix three public keys were stored. This allows the crawler to fully query 14 buckets from every node. The pseudocode in Figure 3.2 summarizes how the node's full local table are parsed. For every bucket a Findnode query is sent for each of the precomputed public keys. The returned neighbors are filtered for duplicates and stored.

```
def parse_node(ID):
    local_table = []
    for i in {0..14}
        pk1, pk2, pk3 = lookup_table[ node-ID XOR mask(i)) ]
        for pk_i in {pk1, pk2, pk3}:
            neighbors = FindNode(pk_i)
            for n in neighbors:
                if n not in local_table:
                    add n to local_table
    return local_table
```

Figure 3.2: Pseudocode for parsing a node's local table

3.2 Querying the network

To query the whole network within a short time, many nodes need to be queried in parallel. Therefore the centerpiece of the crawler is responsible for the orchestration and collection of the results from parsing many nodes in parallel. The crawler makes use of go's primitives *channels* and *syncMaps*. Channels can be thought as FIFO stacks which can be used for communication between go-routines. SyncMaps are dictionaries optimized for concurrent access. The pseudocode in Fig 3.3 explains how these primitives were assembled to query the network efficiently.

To collect ground material the crawler sends Findnode requests to the bootnodes with random public keys. All nodes which need to be queried are added to the queue (implemented as channel). When a worker becomes idle, it takes a node from the queue and checks if it has been parsed by another node since

```
initialize queue
syncMap parsed_nodes = {}

# many workers in parallel
while queue not empty:
    take node from queue
    if node in parsed_nodes:
        continue
    add node to parsed_nodes
    peers = parse_node(node.ID)
    for peer in peers
        if peer not in parsed_node:
            add peer to queue
```

Figure 3.3: Pseudocode describing the parallel execution from one worker's perspective

it was added to the channel. If the node has been parsed by another node it is dropped and a new node is fetched from the queue. If the node is fresh, it is added to the syncMap blocking it for other nodes. Then the worker parses the node as described in 3.2. All discovered peers are added to the queue unless they are found in the dictionary of parsed nodes. Finally, the local table is stored to a file for later analysis.

The output of the crawl is a list of unique enodes, unique meaning the combination of node-ID, IP, port and discport is unique. This implies that the same node-ID can be associated with multiple IP-port combinations or vice-versa. Additionally the list of unique IPs is stores as well as the local table of every node. The crawler is run everyday at midnight and takes approximately 30 minutes to complete.

3.3 Connectivity

The connectivity checker runs two different instances. First is sends a ping over the Ethereum network to all unique enodes discovered in today's crawl. Afterwards an ICMP ping is sent to all unique IPs. The list of responsive enodes and IPs is archived. Tracking the enodes provides information about the connectivity of the nodes while the ICMP ping gives insight into the connectivity of the machines running the nodes. The connectivity checker is run once every hour and takes as input the list of discovered IPs and enodes from the most recent crawl. For the network ping we used the code of Go Ethereum and for the ICMP ping we used the command line tool ping. Since this program is run every hour the code is run in parallel again in order to complete in under 5 minutes.

3.4 Filtering for mainnet nodes

The Ethereum ecosystem consists of many independent networks that follow the Ethereum protocol but are not interacting with each other. In the discovery layer it is not possible to conclude in which network a node is participating. The *mainnet* is Ethereum's production blockchain, this what is referred to when talking about the cryptocurrency. To get a lower bound on the portion of IPs and enodes participating in the actual mainnet the data from our crawler is compared to the data collected by *ethernodes.org* [5].

Chapter 4

Results

4.1 Crawl Insights

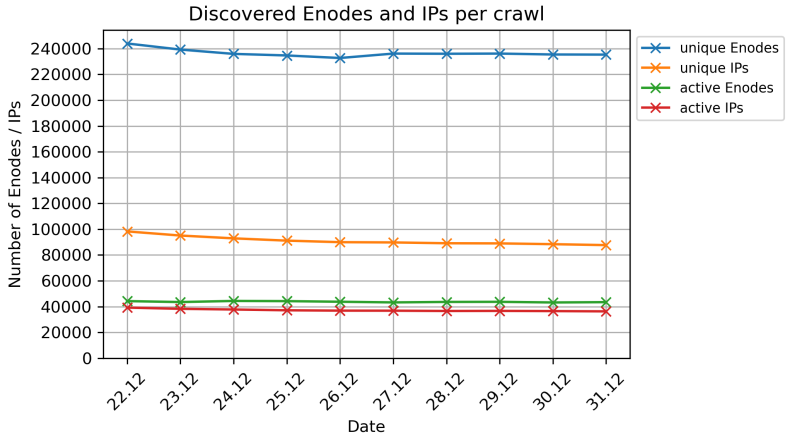

Figure 4.1: Crawl statistics from December 22nd to 29th.

In Fig 4.1 the key figures of the crawls are depicted. The chart's blue and orange line represent the number of *unique Enodes* respectively unique IPs discovered during the crawl. *Unique Enodes* means, as described in section 3.2, the unique combination of node-ID, IP, port and discport (if present). The number of *active Enodes* respectively *active IPs* are visualized by the green respectively blue line. An enode or IP is marked as active if it has been discovered during the crawl and replied to the Ping in the first connectivity check after the crawl.

The chart shows that the number of discovered Enodes is much higher than the number of discovered IPs. By the pigeon hole principle this means that some IPs are used by different node-IDs, these IPs rotate through different enodes. The active enodes to active IPs ratio is close to 1. All four values did not fluctuate significantly during the measurement period. It is noticeable that the difference between active Enodes and IPs is much smaller than the difference between dis-

covered Enodes and IPs. For both the IPs and enodes more than half of the discovered values were inactive. The obvious reason why an enode or IP was unreachable, is that it was shut down. However, it is also possible that they hide behind a NAT which makes those IPs harder to track.

4.2 Stats over Time

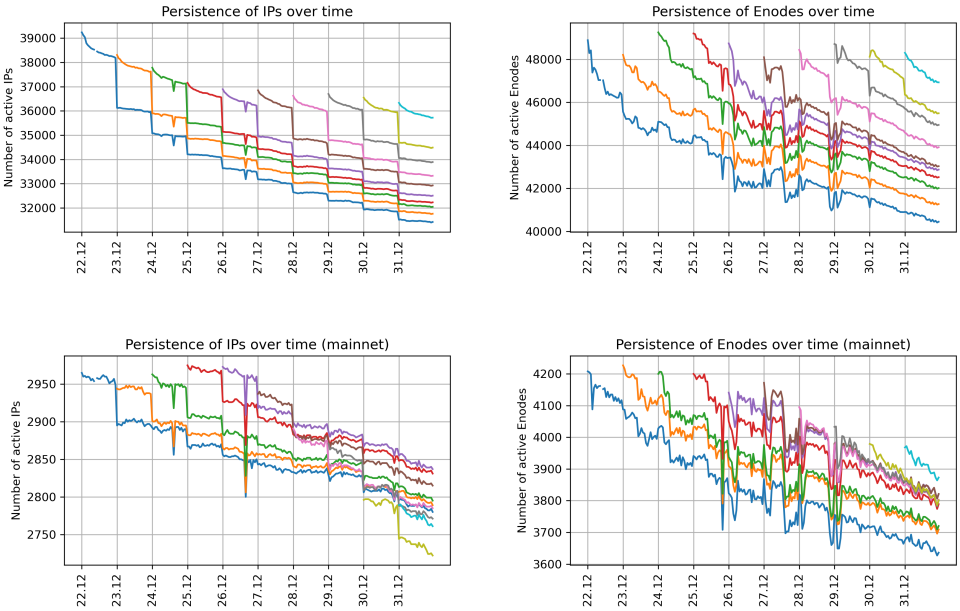

Figure 4.2: Active IPs and enodes over time

The plots in Figure 4.2 visualize the evolution of the active IPs and enodes over time. As described in section 3.3 the IPs and enodes discovered during each of the daily crawls serve as input to the connectivity checks throughout the next day. In 4.2 the list of discovered IPs and Enodes are traced over the days following the crawl and the number of responsive IPs and enodes is plotted.

There is a significant drop of active IPs following each crawl, describing that a number of IPs that responded to the ICMP ping one hour prior to the crawl were not discovered during the crawl. Manual inspection resolved this anomaly. The IPs that drop out every day after the crawl translate to enodes which were inactive for a longer time. The responsiveness of IPs is checked using ICMP ping, which does not allow to specify a port, therefore only the responsiveness of the actual IP is checked but not what service is running behind this IP. Out of all enodes corresponding to the IPs dropping out in the new crawl, only a handful were reachable over the Ethereum network throughout last day. This indicates

that these IPs correspond to 'dead' entries which have not been erased from all routing tables and that the IPs used by enodes are used for multiple services.

The charts in Figure 4.3 describe the overlap between crawls. The overlap measures which percentage of active IPs and enodes from a crawl were discovered future crawls again. The plot shows that 5% to 7% of the active enodes and IPs left the network after one day. After one week the overlap in IPs is close to 85%, while the overlap of active enodes is slightly higher. The IPs and enodes discovered in the mainnet are much more consistent. More than 95% of the IPs and more than 90% of the enodes stayed in the mainnet for more than a week. This suggests that participants of the mainnet stay in the network for longer periods than in the rest of the network.

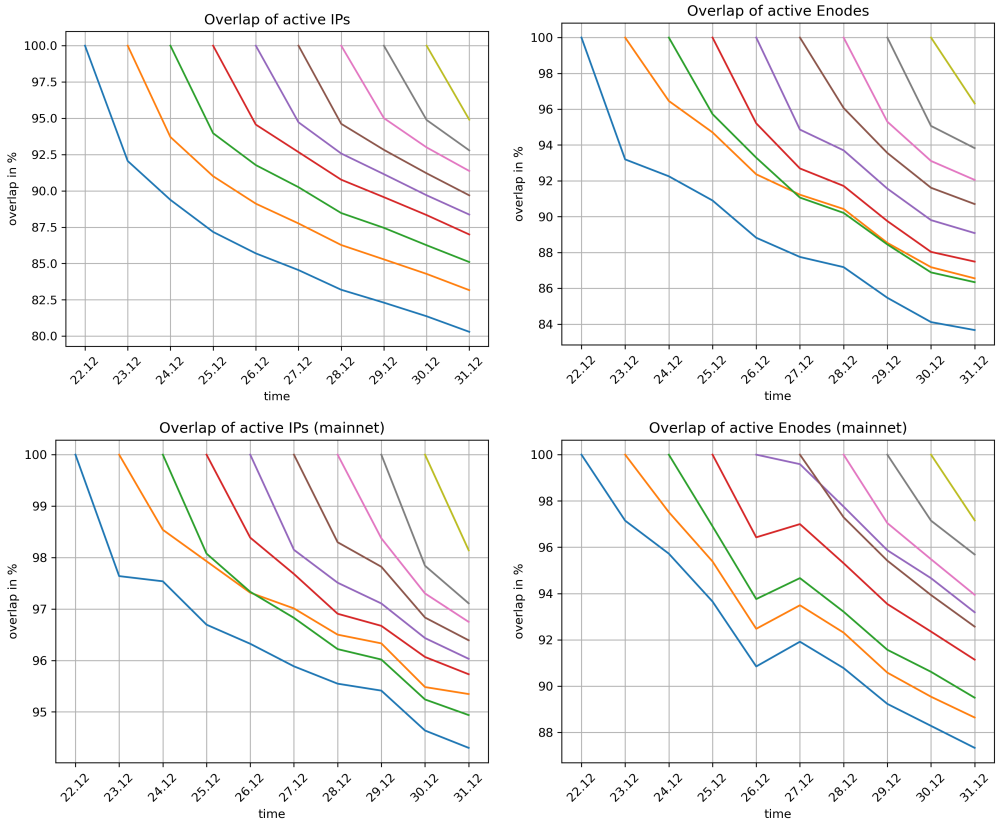

Figure 4.3: IP and enode Overlap over time interval of one week

4.3 Enode Insights

In this section the insights of the enode's local table are presented. Figure 4.4 portrays the distribution of the bucket contents in the format of a boxplot. In

Figure 4.5 the bucket counts of are displayed as a cumulative distribution function. Lastly, the statistics about the local tables are summed up in a histogram in Figure 4.6 counting the total number of entries in the local table of each node.

In Figure 4.4 the orange line represents the median, the boxes visualize the 25%, respectively 75% quartile and the whisker has the length of 1.5 Inter quartile ranges. As expected, with increasing bucket index the number of entries in the local table decreases. For indexes greater than 12 most buckets are empty. For bucket 0 the median lies at 16 peers and the deviation is fairly small. The deviation increases with larger bucket indexes. This described the mode how nodes act when they cannot return 16 nodes from a bucket. In this case entries from buckets with higher index are returned (thus cleaning out the rest of the local table).

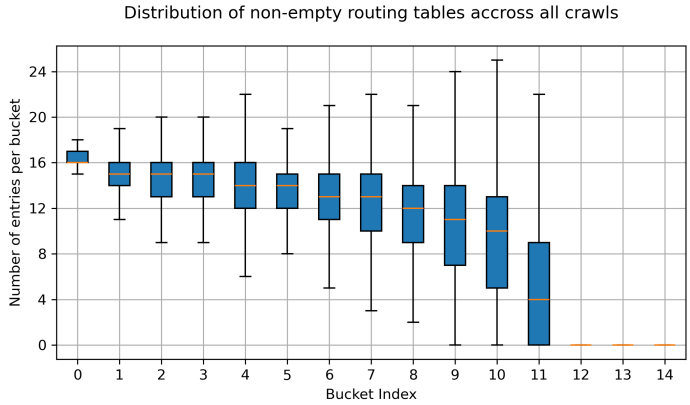

Figure 4.4: Boxplot of the Bucket contents

For each data point in Figure 4.5 the y-value describes the percentage of nodes that returned less neighbors then the number displayed on the x-axis. The main insight from this plot is as in 4.5 the rapid decrease of returned neighbors around bucket 11.

Throughout the measurement period the median of entries in the routing tables was 153 nodes, slightly higher than the mean value of 146.9. In the histogram three peaks can be recognized. A main peak around 160 entries and two minor spikes around 100 entries and 20 entries. The peaks around 160 and 100 can be explained with different configurations. *Go-ethereum* keeps by default 16 entries per bucket plus a replacement cache with 10 entries. While different clients use other values it is also possible to configure *go-ethereum* to use a different value. The peak to the very left of the graphic likely originates from nodes that joined the network very recently before the crawl.

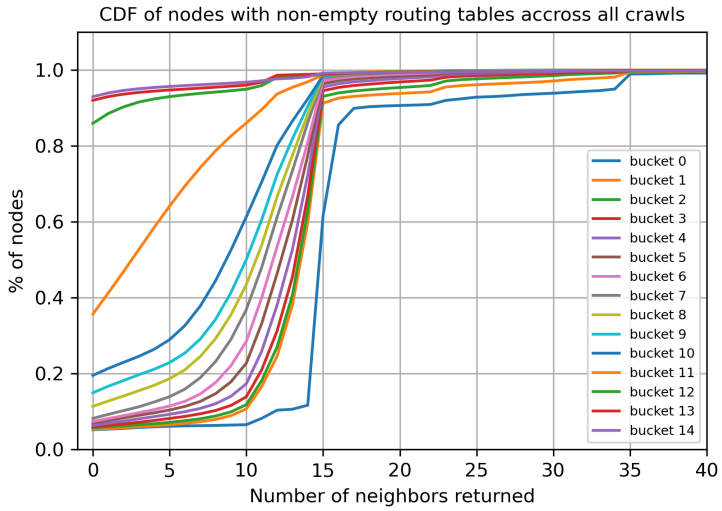

Figure 4.5: CDF of Bucket contents

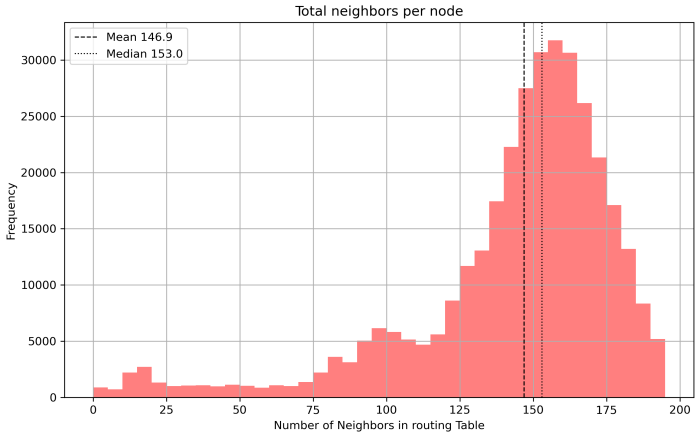

Figure 4.6: total Bucket contents

4.4 Routing table cleanup

In Figure 4.7 it is portrayed how long it takes until entries of IPs that went offline disappear from all routing tables. A random set of IPs was chosen, by the criteria that they were discovered during the crawl of December 27 and went offline during the day and stayed offline for the rest of the measuring period. The plot counts the number of routing tables that carried the corresponding IP a few days before they went offline until the end of the measurement period on a logarithmic scale. A clear trend that the number of routing table entries reduces rapidly as soon as it gets offline, while it can take a long time until the entry is removed from all tables. It is noticeable that a few IPs are erased from the routing tables while their IP was still reachable. Further it is remarkable that it can take quite a while until the entries get erased from all routing tables, in Figure 4.7 it is clearly visible that the inactive enodes entries have not been fully erased from all tables after a week. Likely this can be explained with the replacement strategy of the discovery protocol, inactive entries only get replaced when a new node of the same bucket is encountered. The chance of triggering this replacement mechanism decays with increasing bucket index, thus it is harder for an entry to get replaced in a bucket with a higher index.

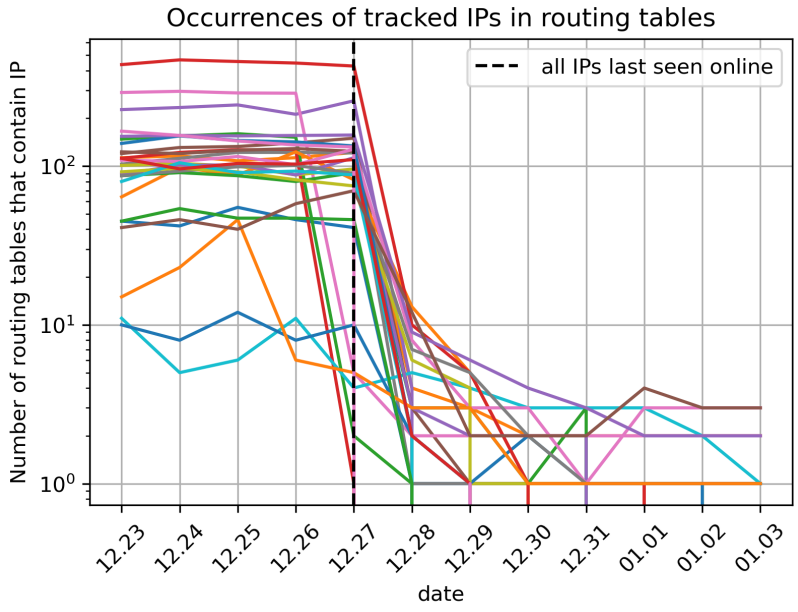

Figure 4.7: Tracking of IPs that went offline

4.5 Comparing Beacon and execution layer

In this section we compare the execution layer to the consensus layer of Ethereum. In Figure 4.8 the orange line describes the number of IPs discovered during our execution layer crawl sole in the execution layer crawl. The blue line describes the number of IPs discovered only in the crawl of the consensus layer. The green line represents the number of IPs that appeared in both crawls. On the right side in Figure 4.8 we plot the same characteristics for the participants identified to take part in the mainnet. The procedure to identify the mainnet participants of the execution layer is described in 3.4, for the consensus layer we filtered the mainnet ENRs by their fork_digest as described by [1].

These graphics have to be interpreted with a grain of salt. The total number of IPs discovered in the execution is in the order of 240′0000 while the consensus layer crawl only holds 30′000 IPs. In the mainnet the opposite holds in the consensus layer roughly 15′000 IPs could be identified as mainnet participants, while the maximum number of mainet participants for the execution layer is around 5′000. Nevertheless, the conclusion can be made that most people participating in the consensus layer also take part in the execution layer.

Figure 4.8: Assigning IPs to consensus and execution layers

Chapter 5

Open Questions

Given the results of this project, the following questions remain open:

- We showed that the discovery protocol as it is used leads to local tables with a lot of dead entries. This might be one of the reasons Ethereum wants to upgrade their discovery protocol to discv5. For further research it would be interesting to measure how much this improves with the upgrade.
- We developed the hypothesis that the latency until these dead entries are replaced in a node's local table increases for higher bucket indexes. Further research could investigate how quickly the old information gets cleaned out of different buckets and how this could be improved.
- The information we could use to identify nodes were the node-ID and the IP. The analysis could be extended up to the TCP-layer which holds much more information about its participants. Enabling to classify which enodes belong to the mainnet on our own, removing the dependency on *ethernodes.org*.
- We did not take nodes behind NATs into consideration. If node is behind a NAT this might impact it's reachability. This topic could be explored by the investigation of enodes that persist in the local tables but are not reachable by the connectivity check.

Bibliography

[1] S. Käser, "Understanding peer-discovery in eth 2.0," Aug. 2023.

[2] Ethereum Foundation. (2023, Jan.) Enode. [Online]. Available: https://ethereum.org/en/developers/docs/networking-layer/network-addresses/#enode

[3] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," 2002.

[4] Ethereum Foundation. (2023, Jul.) Discovery v4. [Online]. Available: https://github.com/ethereum/devp2p/blob/master/discv4.md

[5] Bitfly GmbH. ethernodes.org. [Online]. Available: https://ethernodes.org/

[6] the go-ethereum Authors. (2023, Sep.) Official go implementation of the ethereum protocol eth. [Online]. Available: https://github.com/ethereum/go-ethereum