Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Infrastructure Impact on Cryptocurrency Networks

Semester Thesis

Kyle Fearne

kfearne@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

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

August 20, 2024

Acknowledgements

I would like to thank my project supervisor, Dr Lucianna Kiffer for sharing her invaluable expertise with me, and for her guidance and patience throughout this project. This experience has been very fruitful and has taught me valuable skills that will benefit me throughout the rest of my career.

I would also like to thank the entire Distributed Computing Lab for allowing me to conduct this semester project and for providing all the resources necessary for me to conduct it to the best of my abilities.

Abstract

The aim of this semester project is to analyse any impact that infrastructure could have on the behaviour of cryptocurrencies. The motivation behind this is that recent figures show that a significant number of Ethereum mainnet nodes are being hosted by a handful of major cloud providers, thus undermining a core principle of decentralization. The main contribution of this project was the development of a logging infrastructure that allows for further analysis in the future. The infrastructure that was developed logged messages, peering information and peer-table information from both the Ethereum nodes' Execution and Consensus layers. An automation tool was also developed that allowed the logging to be carried out on multiple machines simultaneously. Finally, a Jupyter Notebook was created as a tool to process and plot the collected data.

From the analysis carried out in this project, some interesting patterns emerged and some location-based biases were detected. There was not enough evidence to state that an Ethereum node running in one of the main cloud provider's datacenter would have any significant advantage when compared to one running on a commercial network, in this case a university network. However, it raises interesting questions for further analysis. This could be done by looking at different parameters than the ones discussed in this project. The tools that were developed will allow for these to be introduced at a later stage.

Contents

Chapter 1

Introduction

1.1 Background & Aims

One of the main promises of cryptocurrencies is removing the dependency on centralized institutions and instead adopting a decentralized model. With this, a distributed network of users can use distributed agreement protocols to bypass the need of a central entity. Such an architecture promotes transparency and accessibility - anyone can set up their own node and participate in this decentralized network. However, recent figures from Ethernodes [1] show that for some cryptocurrencies, a new central point of failure has developed. A significant number of nodes participating in the network are being hosted on major cloud providers such as Amazon AWS and Hetzner. In fact, the cited figures show that around two thirds of all Ethereum mainnet nodes are hosted by just 3 major cloud providers. Of course it is not difficult to see how this may become an issue, especially since it is defying one of the core principles of decentralization by being so reliant on these providers. These providers also gain the ability to infer network topologies, deanonymize communication etc. Potential network-layer attacks that leverage the internet infrastructure itself have been demonstrated and researched in the past [2, 3].

The aim of this project is to analyse the impact of such infrastructure on a cryptocurrency. We hypothesize that running a node out of a datacenter could produce a different behaviour than running a node out of a university network. To do so we aim to analyze peer selection, connection and transaction latencies. The main outcome of this project is to develop a logging infrastructure for the chosen cryptocurrency, as well as to perform some initial analysis of the resulting data.

1.2 Project Structure

This project is divided into three main parts. Chapter 2 discusses the analysis of a number of cryptocurrencies and how one was identified for which to anal-

yse infrastructural impacts. Chapter 3 describes how after selecting a suitable cryptocurrency network, a measurement plan was laid out. A logging client was subsequently set-up such that the measurement plan could be realised. This chapter also discusses the creation of some automated infrastructure that allowed the data collection process to be deployed across multiple machines and to be run at regular intervals. Chapter 4 discusses how the resultant data was processed and plotted, and also presents relevant findings from this project.

1.3 Related Works

Peer-to-peer networks in the context of several cryptocurrencies have been the topic of extensive reseach over the past years. Kiffer et al. present a study [4] about how the Ethereum network evolves over time and how geographic location can alter the peering experience. Kim et al. also present a study [5] which analyzes Ethereum's p2p protocols enabling information propagation and blockchain consensus. This study also discusses the geographic distribution of nodes participating in the Ethereum network.

In a thesis by Hyun-Min Chang [6], node activity of various cryptocurrencies, including their geographic distribution, is discussed. The web-scrapers developed by Chang for this thesis were used in the cryptocurrency selection process discussed in Chapter 2.

Finally, this project also utilized the existing `loggy` submodule added to the go-ethereum client, which was developed by members of the ETH Distributing Computing group. Messages received or sent by the client were recorded into .jsonl format. This is discussed in further detail in Chapter 3.2 This submodule was also used in spyprysm, a modified version of the Prysm Ethereum client, to log consensus layer activity. For spyprysm, this module logged certain activity (discussed further in Chapter 3.3) into .csv files and then zipped these.

Chapter 2

Cryptocurrency Selection

2.1 Overview

The initial part of this semester project dealt with selecting a suitable cryptocurrency for which to analyse infrastructure impacts on. The chosen cryptocurrency had to meet certain criteria - firstly that there were a substantial amount of nodes participating in the network and more importantly, that a significant amout of these were being hosted in an Amazon AWS datacenter. Without the latter it would not be possible to investigate any potential bias that arises from having a significant chunk of the network hosted by one provider. Ethereum (ETH), Ethereum Classic (ETC), Dogecoin (DOGE), Bitcoin Cash (BCH), ZCash (ZEC), Dash (DASH) and Groestlcoin (GRS) were all shortlisted as potential candidates for the analysis.

2.2 Selection Process

A web-scraper was used in order to obtain a list of all active nodes in a particular cryptocurrency network over 24 hours, as well as the information pertaining to those nodes - such as node ID, IP, client information etc. The scraper utilised APIs from a variety of blockchain explorers such as blockchair.com and etcnodes.org [7, 8]. The data was output in the form of csv files, so a simple python script was written in order to parse the scraper's output. The parser takes the nodes' IP, runs a reverse DNS query in order to find the host, and then outputs how many of the nodes are being hosted on Amazon AWS. The parser was run on three days worth of data in order to obtain an average number of active nodes and how many of these were hosted in Amazon AWS. This is detailed in the below Table 2.1

	Number of Nodes	Nodes in Amazon AWS	Percentage in AWS
Ethereum	6105	910	14.9 %
Ethereum Classic	10157	87	0.85 %
Bitcoin Cash	697	82	11 %
Dash	3477	30	0.86 %
Dogecoin	710	46	6.5 %
Litecoin	931	58	6.2 %
ZCash	3485	26	0.75 %
Groestlcoin	47	1	2.14 %

Table 2.1: Nodes for each cryptocurrency

After analyzing the share of nodes that are hosted in Amazon AWS for each of the above cryptocurrencies, it was decided that the Ethereum network was going to be investigated in this project. The above results show that roughly 15% of nodes in the Ethereum network are being run out of an Amazon AWS Datacenter. For a network of this size, this share of nodes is significant and such a high degree of dependence on a centralized party can have a negative impact on the system.

Chapter 3

Logging Clients & Experiments

3.1 Overview

Once Ethereum was chosen as the cryptocurrency to be analysed, it was necessary to develop a logging infrastructure and an experimental plan for how to record data. This chapter describes how this was done. Ever since The Merge in September 2022 [9] where Ethereum switched from proof-of-work to proof-of-stake based consensus, one requires an Execution Client as well as a Consensus Client in order to run an Ethereum node. For the purpose of this project, go-ethereum [10] was chosen as the execution client since it is one of the most widely used client. Prysm [11] was used as the consensus client, it is also written in Go and is also one of the most-widely used Consensus clients [12].

3.2 spygeth

Spygeth is a modified version of the go-ethereum client designed to log messages sent and received by the Ethereum node. An older version of this was already in use by the Distributed Computing Lab, written for an earlier release of the go-ethereum client. This utilized the previously discussed loggy submodule in order to log messages (both in/outbound) defined in the ETH Wire Protocol [13]. These included:

- GetBlockHeadersMsg
- BlockHeadersMsg
- GetBlockBodiesMsg
- BlockBodiesMsg
- GetNodeDataMsg
- NodeDataMsg
- GetReceiptsMsg
- ReceiptsMsg
- NewBlockHashesMsg
- NewBlockMsg
- TxMsg

Firstly, the existing loggy code was adapted to be compatible with the latest release of go-ethereum, version 1.14.7. Logging for the aforementioned messages was maintained, with some changes. For instance, NewBlockMsg and NewBlockHashesMsg were removed from the 'eth' protocol due to block propagation no longer taking place in the execution network after The Merge [14], therefore the logging for these messages was also removed. Logging for GetPooledTransactionsMsg and PooledTransactions was added - recording only a list of the transaction hashes rather than the full message to avoid having huge log files.

The newer version of spygeth implemented in this project includes logging for peering messages as well, to enable tracking which peers a client connects to, duration of said connections as well as further information pertaining to how the peer table is populated and maintained. This activity happens inside the `p2p/server.go` module. To track active connections, the following logging messages were added:

- AddPeer - Whenever an addPeer checkpoint occurred during the execution of the server subroutine, this was logged as a peer being added. This checkpoint occurs after the protocol handshake has been completed successfully, therefore the client will definitely connect to the peers being logged. The peer's IP, node ID, running peercount, current time, and direction of peer (in/outbound) are all logged.
- RemovePeer - Whenever a 'delpeer' checkpoint occurred during the execution of the server subroutine, this was logged as a peer disconnection. The peer's IP, node ID, running peercount, current time and direction of peer (in/outbound), connection duration and reason for disconnection are all logged.

This version of spygeth also introduces logging for peer table information. As part of its node discovery protocol which runs over UDP, go-ethereum makes use of Kademlia which implements a Distributed Hash Table [15]. This table allows the client to keep track of known nodes. The relevant code that maintains this DHT is found inside the `p2p/discover/table.go` module, where the following logging messages were added to track peer table information:

- addSeenNode - In this function, a node which may or may not be live is added to the end of a bucket in the DHT. If there is no space in the bucket, the node is added to a list of replacements to be swapped into the bucket when space comes up. Our logger will only log the node being added once it is actually added into the bucket. The nodes' ID, IP and timestamp are all logged.
- addVerifiedNode - This function adds a node whose existence has been verified to the bucket. If it is already in the bucket it is moved to the front

of the bucket, and if there is no space it is added to the replacement list as explained above. The nodes' ID, IP and timestamp are all logged.

- Node Added (replacement) - This function takes a node from the replacement list and adds it to one of the DHT's bucket. The node is then removed from the replacement list. The nodes' ID, IP and timestamp are all logged.
- Node Deleted - In the `deleteInBucket` function, a node is removed from a DHT's bucket. A check is made before this to ascertain that the node is actually in the DHT before attempting to delete it, to prevent the delete function from being called multiple times on the same node. The nodes' ID, IP and timestamp are all logged.

This latest version of spygeth has been dockerized and pushed to GitLab where it can now be accessed [16] and easily used.

3.3 spyprysm

Similar to spygeth, spyprysm is a modified prysm client that operates at the consensus layer of the Ethereum node. Prior to this project, an earlier version of spyprysm was already in use by the Distributed Computing Lab. This logged attestations, gossip logs (how messages propagate through the network), mesh information and peer information (connection state of the peer). Further information about what was recorded can be found in Appendix A.

Once again, certain changes had to be made to this existing code in order to also start logging peer table information for the consensus layer. Prysm uses DiscoveryV5 [17] in its peer discovery protocol, which is very similar to DiscoveryV4 used in go-ethereum. Peer table maintenance is done inside the `p2p/status.go` submodule. The following were logged:

- Deletions - The `prune` function sorts the entries inside the DHT's buckets by a scoring system. Peers that are outdated or disconnected peers have higher scores, and the pruning starts in order of descending score. When a peer is removed from a bucket, the associated peer information as well as the timestamp are logged.
- Additions - The addition of a node to the peer table is done in the `addIpToTracker` function. Whenever an addition happens, the associated peer information is logged together with the timestamp.

Apart from this, changes had to be made to the `loggy` submodule. A function `LogPeerTableAction` was written to write logs of the peer table changes into a .csv file.

3.4 Experimental Plan & Set-up

This section discusses the set-up of machines for the experimentation conducted, as well as the plan on how the data recording took place.

The plan was to have five Ethereum nodes running on five different machines. The first node was run on an ETH machine in Zurich. To test the impact of running nodes on different providers, a further two nodes were run in close proximity to each other - one on a Hetzner machine in Falkenstein and another on AWS in Frankfurt. Finally, the impact of geographical location on the nodes' behaviour was of interest, so a further two nodes were run on AWS machines, but this time based in Seoul and Virginia. All of these machines had 8 cores and 32GB of memory, except for the ETH Machine which had 20 cores and 126GB of memory. For the Ethereum nodes to be run, both the spygeth and spyprysm containers were launched.

In order to have sufficient data to analyze, 6 data-logging runs were conducted across six days - each of which running from 06:00 UTC to 22:00 UTC. During the first three days, the machines had their UDP and TCP ports closed, whilst for the last three days these were opened. The idea here was to be able to analyze what effect this had with regards to peering and how quickly messages were propagated to all five Ethereum nodes. The node running on the ETH machine in Zurich was behind a NAT, therefore the ports remained closed for all six days. The ports in question were:

- 30303 UDP/TCP - These are the ports used by the go-ethereum client for peer-to-peer communication. The TCP port is the client's listener port whilst the UDP port is used by the node discovery protocol.
- 13000 TCP - This is used by the prysm client for peer connectivity. Gossip/p2p communication also flows through this port.
- 12000 UDP - This is used by the prysm client for node discovery and chain data.

Resource usage on each of the machine was also recorded through `resources.py`. This script created a log file that recorded resource usage metrics every minute. The log included:

- CPU Usage
- Virtual Memory Usage
- Swap Usage
- Bandwidth (In)
- Bandwidth (Out)
- Timestamp

3.5 Automation Infrastructure

To facilitate the data logging process, automation infrastructure was created. This enables recreation of this logging process in the future. The script `log_script.py` was scheduled to run on every machine at 06:00 UTC using `cron`. The script takes the duration (in seconds) of the run that a user requires as a parameter and does the following:

1. The folders containing the logs for spygeth and spyprysm are deleted. These folders are cleared at the start so that they only contain logs for one run of data-collection.
2. The `nodes` and `nodekey` are folder deleted from go-ethereum's data directory. This is done to clear the client's peertable and to ensure that the client has a fresh enode ID at the start of each run. The enode ID is what identifies a node, so this ensures having freshness in the experiments and that no nodes in the network are already aware of the Ethereum node being run.
3. A directory tree is created which will hold all the logs pertaining to that run. The directory tree created is as follows:

```
Logs_[starting_time]
├── geth
├── prysm
└── about
```

4. Spygeth and spyprysm are both launched. Their output is recorded in `geth_output.txt` and `spyprysm_output.txt` respectively. Any exceptions are logged in `geth_stderr.txt` and `spyprysm_stderr.txt`.
5. `resources.py` is also launched which creates a `ResourceLog.json` file inside the parent log directory.
6. The script then sleeps for the duration specified by the script parameter. Logging is ongoing during this time.
7. Node information for both the spygeth and spyprysm clients is obtained. For spygeth, an RPC call is made to port 8545, whilst an HTTP GET request is made to port 3500 for spyprysm. This information is logged in the `/about` subdirectory.
8. Spygeth, spyprysm and the resource script are all terminated. All the logs are copied over from the logging directories to the directory tree created in step 3 such that all information pertaining to the run is now inside this directory structure.

Chapter 4

Results & Plots

This chapter aims to analyse the data obtained in the data-logging phase and attempts to find any evidence of infrastructure having an impact on the Ethereum nodes. The data is processed and plotted using a Jupyter Notebook written for this project.

4.1 Resource Usage

Firstly, the `ResourceLog.json` files were analysed. These files gave information about the resources used by each machine. Figure 4.1 below shows the CPU usage of the machines running with closed and open ports. The plot for ws203 was omitted in this case, since the machine's specs are different from the rest, as previously mentioned. This figure shows that machines hosted in Amazon AWS had an average CPU usage of around 8%. It can be seen that in general the CPU usage seems to be slightly higher across all machines with open ports.

Next, bandwidth usage for each machine was plotted. Figure 4.2 shows the inbound and outbound bandwidth metrics when the ports were closed whilst Figure 4.3 shows shows the same metrics when the ports are opened. Similar to the CPU plots, it can be seen that the average bandwidth usage is slightly higher when the ports are opened. Perhaps more interestingly, Figure 4.4 and Figure 4.5 show how this bandwidth usage scales with the number of peers the spygeth client connects to. An important point to note is how the peercount changes when the ports are opened. With closed ports, we only have outgoing connections limited at 16 peers. Once the ports are opened incoming connections can also be made, and hence the ceiling of connections rises to 50 peers.

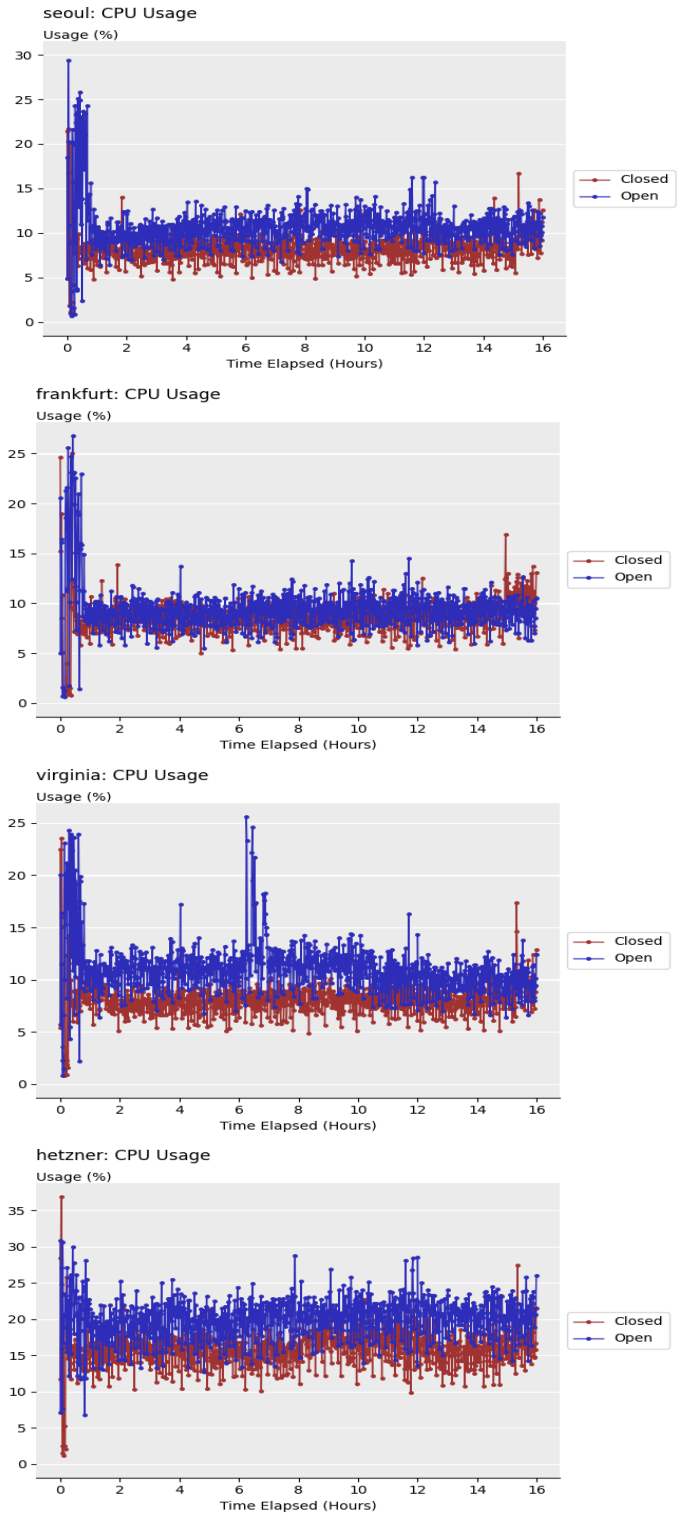


Figure 4.1: CPU Usage

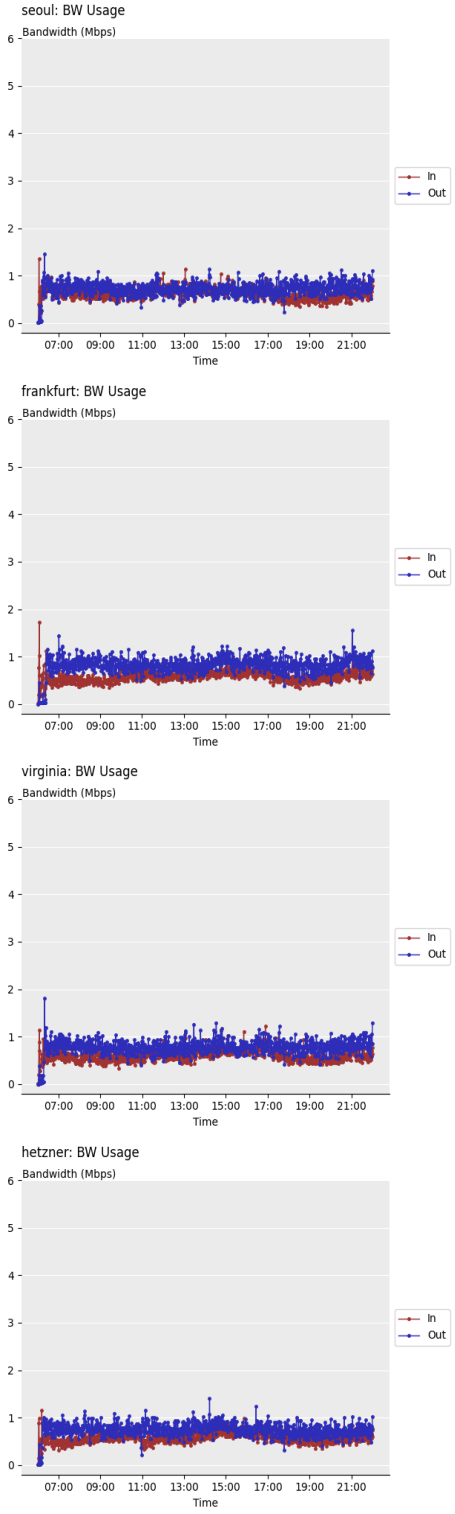


Figure 4.2: Bandwidth usage with Closed ports.

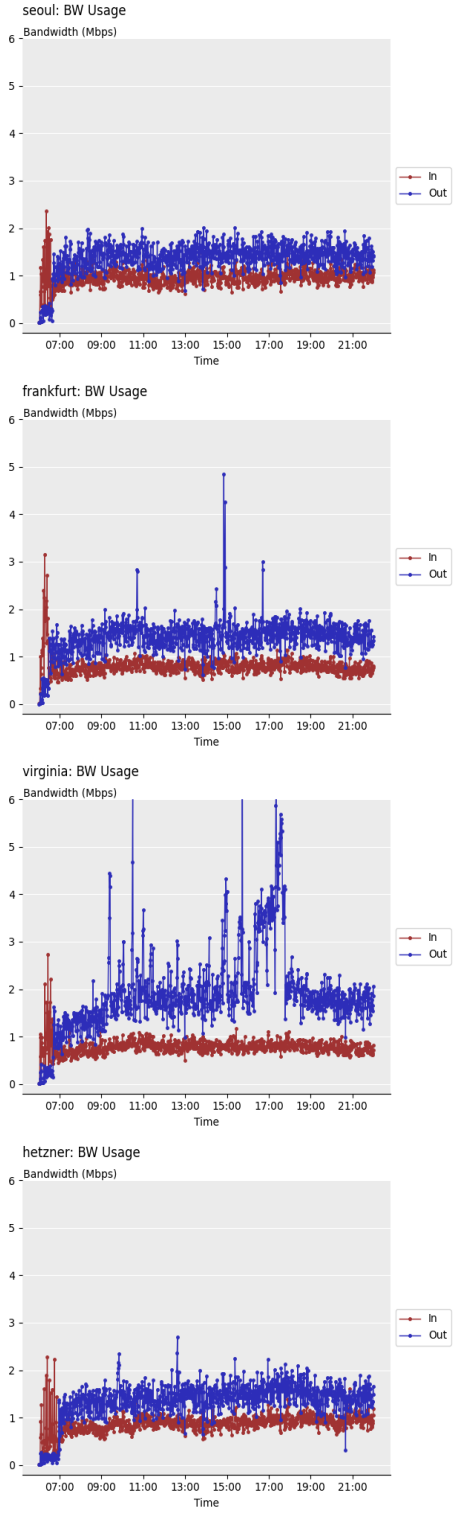


Figure 4.3: Bandwidth usage with Open ports.

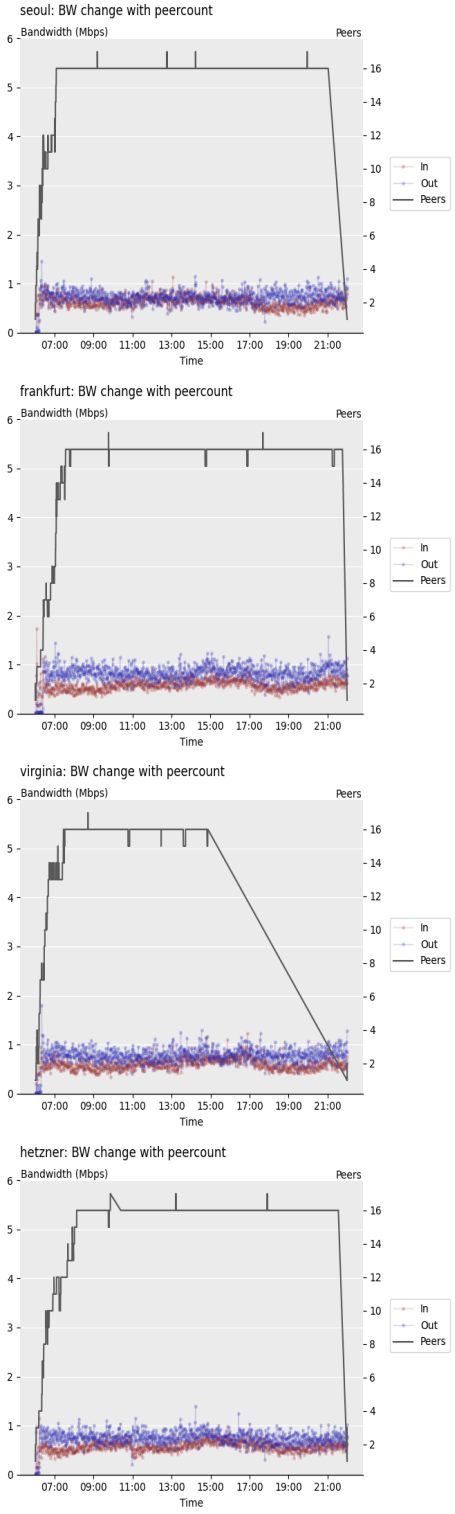


Figure 4.4: Bandwidth scaling with peer-count (Closed)

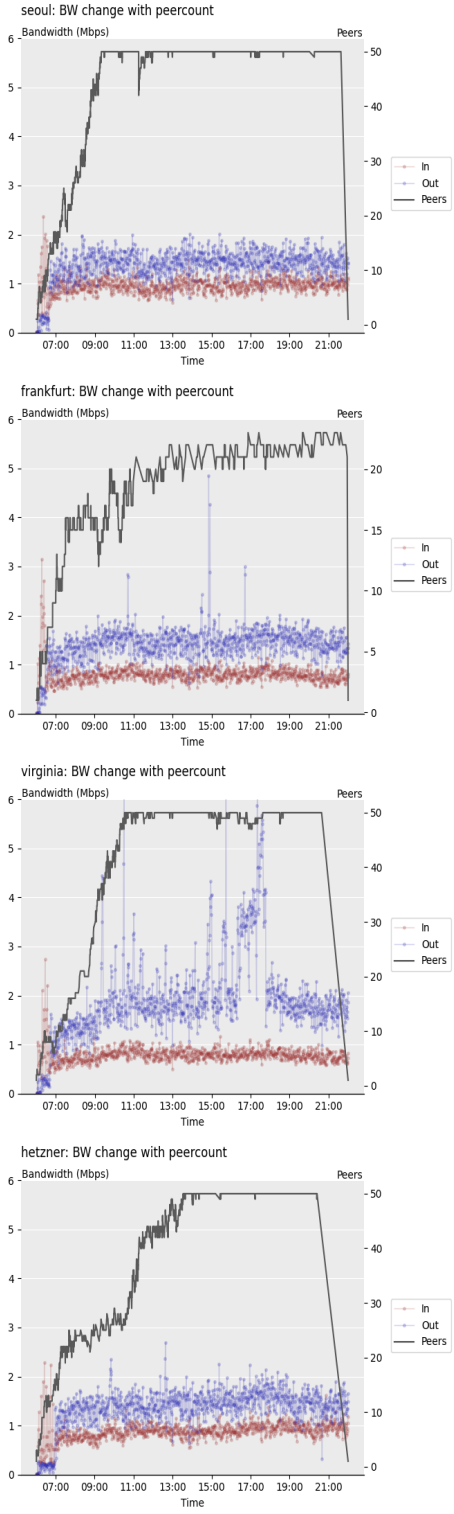


Figure 4.5: Bandwidth scaling with peer-count (Open)

4.2 Peer Connectivity

This next section presents some insights into the data obtained from the peer-to-peer logs, recently added to spygeth and spyprysm as discussed in Sections 3.2 and 3.3.

Starting with Figure 4.6, a CDF plot was made for the duration of connections made by the spygeth client. The data seen in these plots comes from all the six days of data collection. This means that the data from the three runs of open/closed ports were aggregated in the same plot, to get an idea of the average behaviour. The dotted vertical lines represent the average connection duration. It is important to note that the x-axis is a log-scale, and also that between closed and open ports, the number of active peers changes from ≈ 16 to ≈ 50 as demonstrated previously. This means that there are a lot more attempted connections when the ports are opened. Initially, this fact lead to the belief that the average connection duration would be shorter when the ports were opened due to a lot of futile, unstable connections. However, the plots show that in all but one machine the average connection duration is longer with open ports. This could possibly be due to having more peers to choose from, thus enabling the choice of more stable peers. Since machine ws203 is behind a NAT and the opening of ports made no material difference, we see that the behaviour in the first three and the last three days was roughly the same.

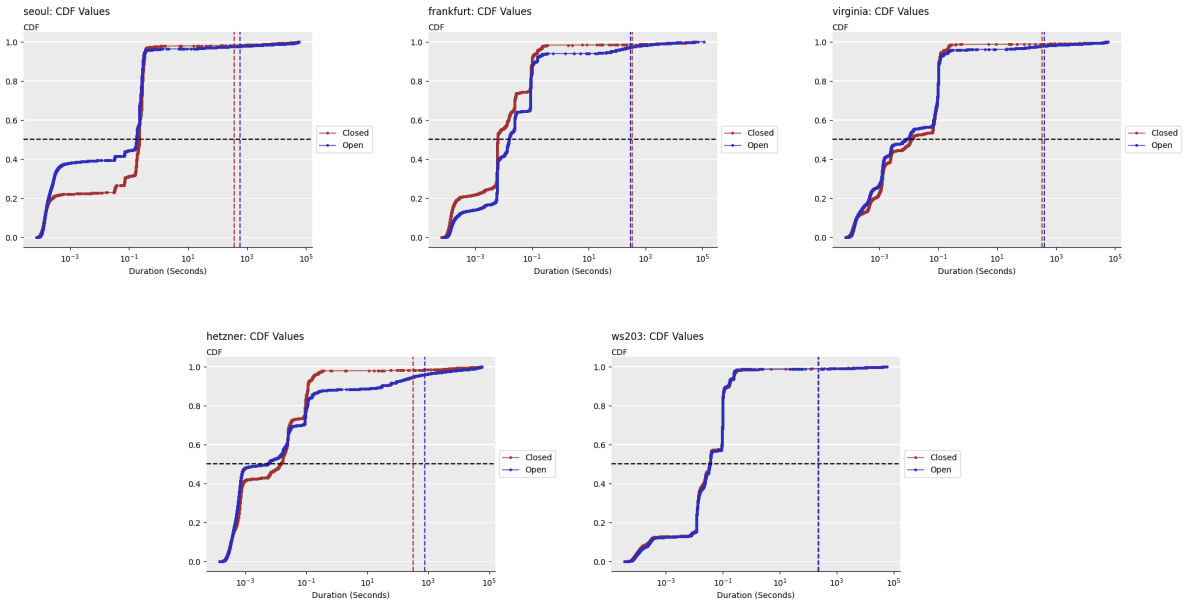


Figure 4.6: CDF of Connection Durations

With these figures now available, it was of interest to investigate where spygeth's most stable peers were located. To do this, the top 50 longest connections (with open ports) were analysed. A reverse DNS lookup was done in order to get the provider, and the `ip2geotools` [18] library was used to get the peers' geographic location. Figure 4.7 and Figure 4.8 show the distribution of the most stable peers by country and continent respectively. These are also grouped by peers hosted on Amazon and those that are not.

It was observed that for all the machines, most of the stable peers were based in Europe. The countries where most of our machines' stable peers were located were the US and Germany, which made sense when considering the general distribution of Ethereum Mainnet nodes. Ethernodes Statistics [1] state that these two countries are the two countries hosting the most nodes. Most of the stable peers that were also being hosted on Amazon AWS were based in North America.

Since in this project the peer table entries were also logged for spygeth and spyprysm, it was possible to plot how these grew over time during the experiment runs. Figure 4.9 and Figure 4.10 show the peer table growth for closed and open ports respectively. A point to note was that the size of the peer table plateaued at 200 on the plots, indicating its capacity. Interestingly, this was not always the case for all machines. For instance, on the Virginia machine, the number of peers in the peertable seems to hover around the 160 mark. This was also noted for the Frankfurt machine.

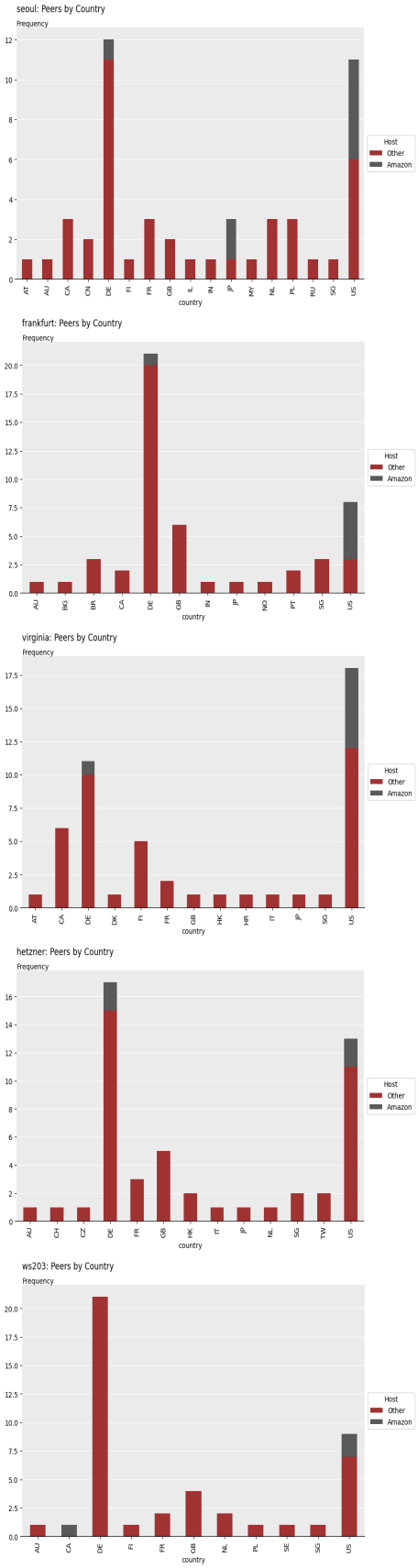


Figure 4.7: Peer Distribution by Country

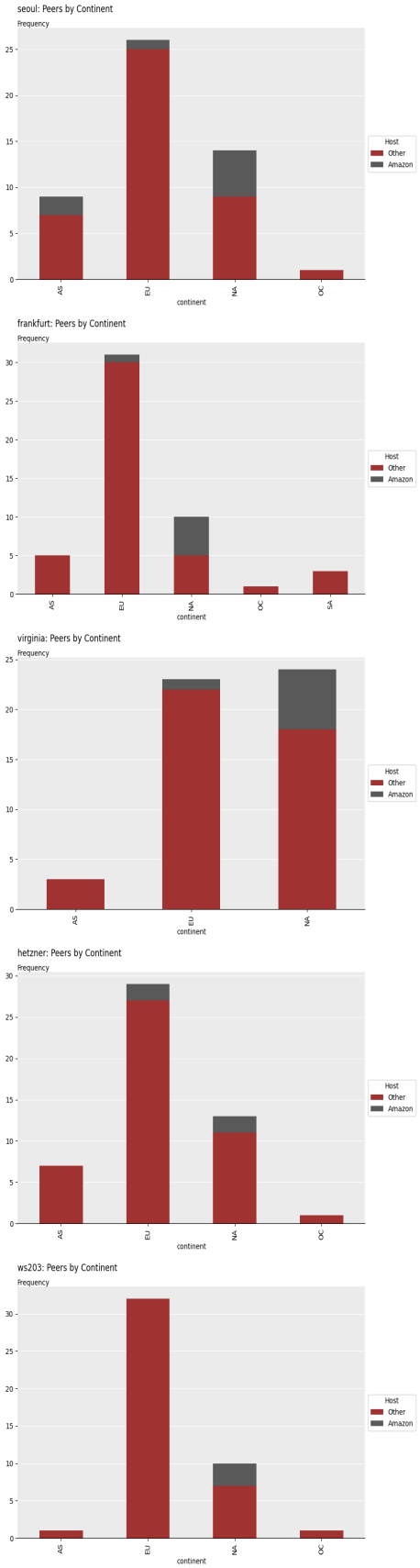
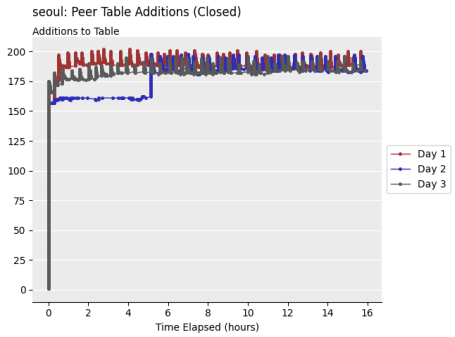
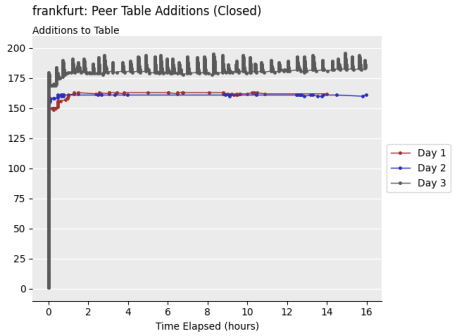
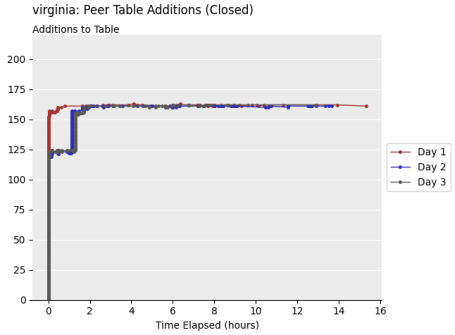
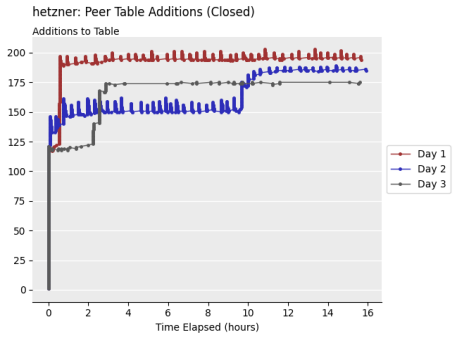


Figure 4.8: Peer Distribution by Continent









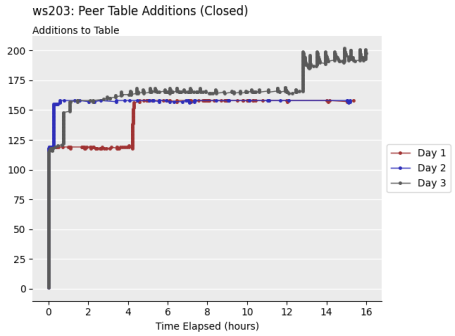
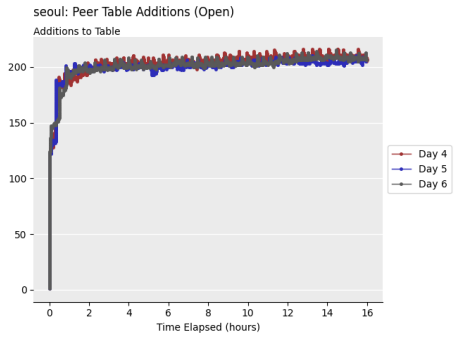
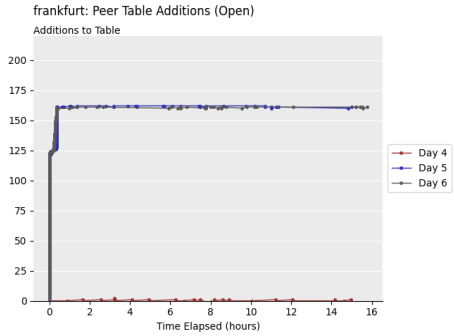
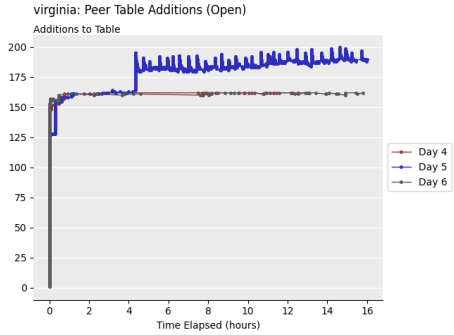
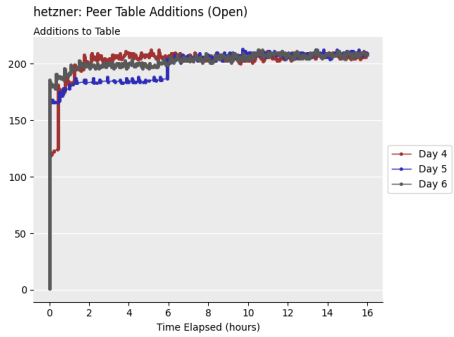


Figure 4.9: Peertable Size (Closed)









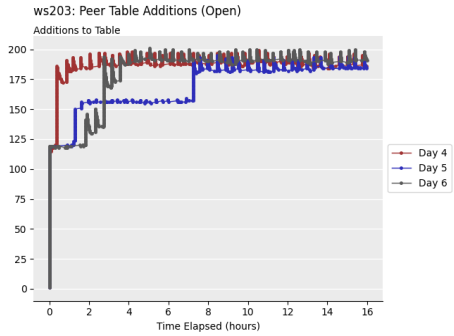


Figure 4.10: Peertable Size (Open)

4.3 Message Delays

Apart from the peering behaviour, the project's analysis of infrastructure impact also aims to look at message delays and propagation. Over the course of one run, many transactions are received by each machine. A subset of these transactions are received by all five machines in the experiment. The table below shows how many transactions each machine received on average during one run.

Each machine's transaction list was sorted in ascending order by time received and then duplicates were removed, keeping only the first instance. Next, an inner join on the transaction hash was done to find the transactions received by all machines. All timestamps were converted to UTC, and then the earliest timestamp was found. The bar chart in Figure 4.11 shows how often each machine received a particular transaction first. Following this, the earliest timestamp was subtracted from each machine's own timestamp for that transaction in order to find the delay in receiving the message. Initially, all delays were plotted however this caused extreme skewing as some machines received a message hours later. To combat this, an upper threshold of 1 minute was chosen, beyond which that particular message was excluded. Anything below 1 minute was considered delay due to the network, so was relevant to this analysis. As an example, on the last day the number of transactions received were as follows:

Machine	No. of Txs
Seoul	3,585,084
Frankfurt	1,289,210
Virginia	1,807,612
Hetzner	2,328,351
ws203	890,392

Table 4.1: Transaction Count

Table 4.1 already highlighted some differences between the machines. For instance, the table showed that ws203 was at a disadvantage. The Seoul node received 53% more transactions than the Hetzner node which had the next highest number of transactions.

After the join was performed, there were only a total of 271,057 transactions that were seen by all the machines. Out of these, 166,440 transactions have a delay of less than one minute across all machines. From Figure 4.11 it can be observed that from these transactions, the Hetzner node was the first machine to receive a transaction the most times and the Seoul node was the first the least amount of times. The latter was particularly interesting given the fact that the Seoul node received the most transactions. After this, the average distribution of delays was plotted in a box-plot, as shown in Figure 4.12. The black triangular

shape represented the mean delay (0.07s for Hetzner - 0.14s for Seoul). Hetzner and ws203 showed the highest degree of variance, whilst Seoul showed the least. Since Hetzner and ws203 had the highest amount of transactions that they received first, their lower quartile is at 0 (no delay). It was observed that the three European machines were the top three in terms of how often they were the first to receive a transaction. This was not surprising due to the fact that only the transactions present in all five machines were considered. The three European machines probably skew this overlap of transactions as they see a lot of the same transactions.

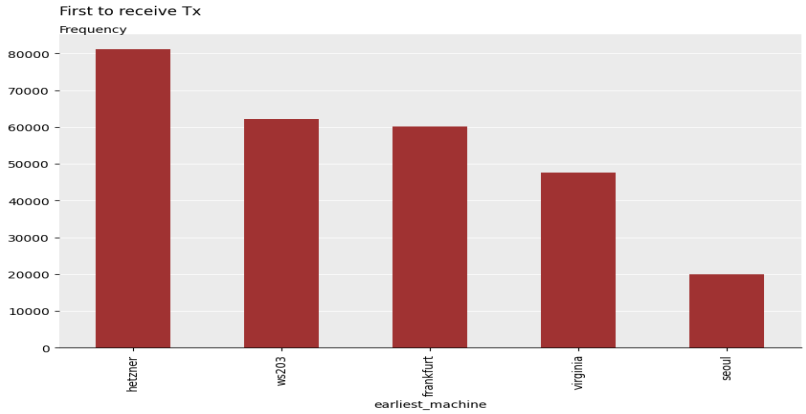


Figure 4.11: Frequency of first received Transactions

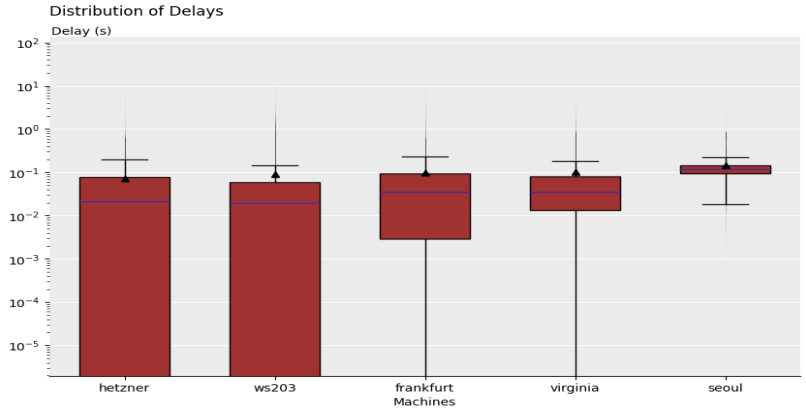


Figure 4.12: Distribution of Transaction Delays

The same analysis was also carried out on the consensus layer, looking at attestation aggregations received by the spyprysm client. Once again, the total amount of attestation aggregates received by each machine were as follow:

There were a total of 4304 messages that were present on all five machines.

Machine	Attestation Aggregates Received	Unique
Seoul	8,919,325	5083
Frankfurt	8,344,048	4966
Virginia	9,012,566	5054
Hetzner	8,412,554	4902
ws203	6,675,095	4311

Table 4.2: Attestation Aggregate Count

This time, the Frankfurt node was the best performing and the Seoul node was once again the first node the least times. In general, the mean delay times are shorter in the analysis of the consensus layer than the previously discussed delays for transactions. The delay distribution can be seen in Figure 4.14. Once again, the top three machines that were first to receive a particular AttestationAggregate most often were the European machines. This showed that there was a bit of an advantage that these machines had over the others.

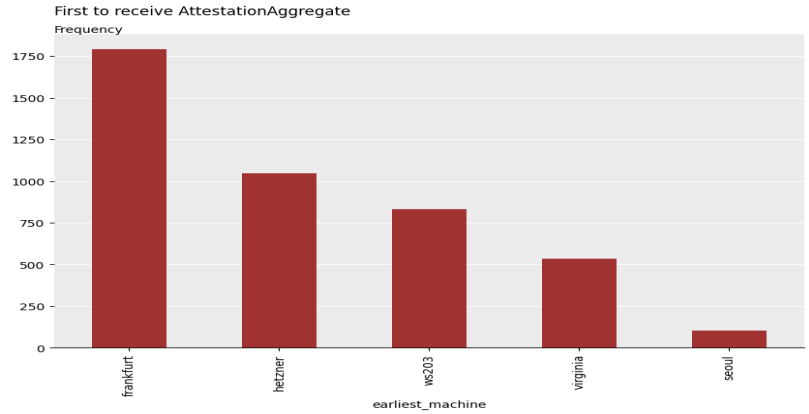


Figure 4.13: Frequency of first received Attestation Aggregate

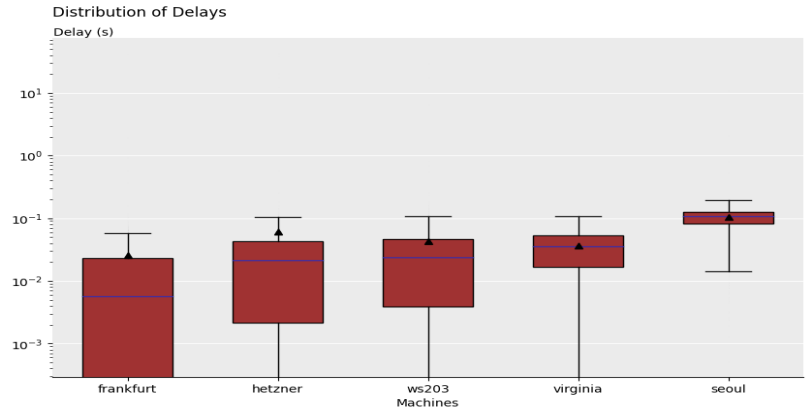


Figure 4.14: Distribution of delays

4.4 Subnet Plots

Finally, the mesh sizes per subnet in the Consensus Layer were plotted.

The plots in Figure 4.15 show how the sizes of the Beacon Chain subnets varied across the whole experiment. It is worth recalling at this point that the first three days of the experiment were run with closed ports, whilst during the last days the ports were opened. It could be seen that the mesh sizes actually increased across all machines once the ports were opened, with the notable exception of ws203 that was behind a NAT. At any given time, each machine was subscribed to only two subnets picked randomly. Over the course of the experiment, these two subnets changed as can also be seen in the plots. The data is not continuous as the experiments were not running 24 hours daily, however, in some points the plot shows that the same subnets could have been selected on multiple days, thus giving the appearance of a continuation in the plot. Note that there is a missing day of data for the Zurich machine on the 19th of July. This was due to an incorrectly configured automation script which was resolved for the next run.

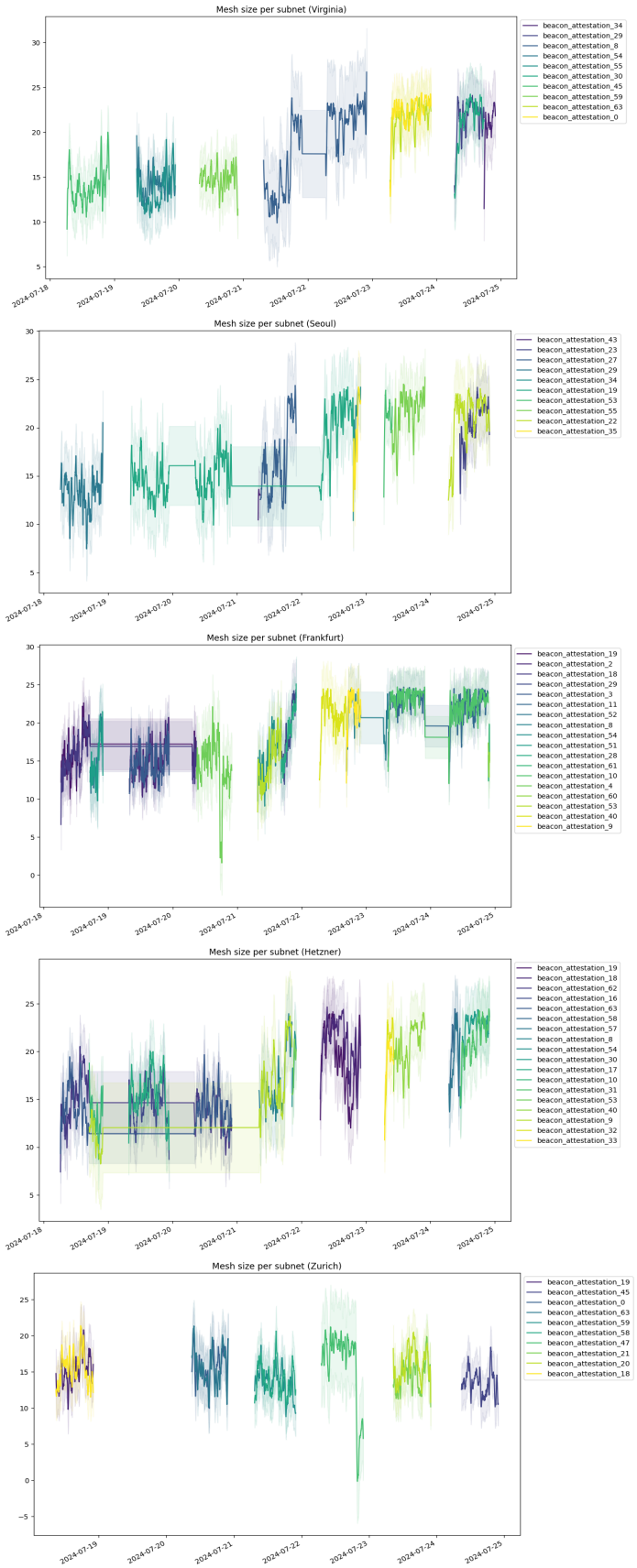


Figure 4.15: Mesh size per subnet

Chapter 5

Conclusion

An infrastructure was developed to enable the logging of messages and p2p information for multiple Ethereum Nodes spread across different machines. This was done through the modified execution and consensus clients - spygeth and spyprysm. Both of these are available as Docker containers for easy use by anyone who wishes to conduct further experimentation in the future.

A Jupyter Notebook was also written in order to process the data and output the plots related to resource usage, peercounts, peer-table information, message delays and more as discussed in Chapter 4.

Overall, some interesting patterns in the behaviour of nodes were observed. There were also some location-based biases which were detected. The preliminary analyses presented in this project does not show a clear bias in running nodes out of cloud providers, however it raises interesting questions for further investigation. Importantly, the work carried out on the infrastructure will allow for further analysis on different metrics where it might be possible to identify a more concrete bias experienced by nodes running on major cloud providers.

Bibliography

[1] "Ethereum mainnet statistics," https://ethernodes.org/networkType/Hosting, accessed: 2024-08-10.

[2] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking bitcoin: Routing attacks on cryptocurrencies," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 375–392.

[3] M. Apostolaki, C. Maire, and L. Vanbever, "Perimeter: A network-layer attack on the anonymity of cryptocurrencies," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 147–166. [Online]. Available: https://doi.org/10.1007/978-3-662-64322-8_7

[4] L. Kiffer, A. Salman, D. Levin, A. Mislove, and C. Nita-Rotaru, "Under the hood of the ethereum gossip protocol," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 437–456. [Online]. Available: https://doi.org/10.1007/978-3-662-64331-0_23

[5] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey, "Measuring ethereum network peers," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 91–104. [Online]. Available: https://doi.org/10.1145/3278532.3278542

[6] H.-M. Chang, "Measuring cryptocurrency networks," Bachelor's Thesis, ETH Zürich, 2023.

[7] "Blockchain explorer, analytics and web services," blockchair.com, accessed: 2024-08-07.

[8] "Etc node explorer," etcnodes.org, accessed: 2024-08-07.

[9] "Ethereum roadmap: The merge," https://ethereum.org/en/roadmap/merge/, accessed: 2024-08-10.

[10] "go-ethereum docs," https://geth.ethereum.org/docs.org, accessed: 2024-08-07.

[11] "Prysm: Github repository," https://github.com/prysmaticlabs/prysm, accessed: 2024-08-07.

[12] "Ethereum cl network public dashboard," https://monitoreth.io/nodes, accessed: 2024-08-10.

[13] "Ethereum docs: Wire subprotocol," https://ethereum.org/en/developers/docs/networking-layer/#wire-protocol, accessed: 2024-08-07.

[14] "Eip-3675: Upgrade consensus to proof-of-stake - specification of the consensus mechanism upgrade on ethereum mainnet that introduces proof-of-stake," https://eips.ethereum.org/EIPS/eip-3675#network, accessed: 2024-08-07.

[15] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. Berlin, Heidelberg: Springer-Verlag, 2002, p. 53–65.

[16] "spygeth: Gitlab repository," https://gitlab.ethz.ch/disco-students/fs24/fearne-spy-geth-and-prysm, accessed: 2024-08-07.

[17] "Prysm docs: Discoveryv5," https://docs.prylabs.network/docs/devtools/net-design#discoveryv5, accessed: 2024-08-10.

[18] "Ip2geotools python library," https://pypi.org/project/ip2geotools/, accessed: 2024-08-10.

Appendix A

Full List of Logged Items

A.1 Execution Client

A.1.1 Info

- Dictionary received from RPC Call. Name, enode, IP etc of execution client.

A.1.2 TxMsg

- TxHash - Transaction Hash
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.3 GetBlockHeadersMsg

- Message - Includes Request ID, Origin (Hash, Number), Amount, Reverse (t/f)
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.4 GetBlockBodiesMsg

- Request ID
- GetBlockBodiesRequest - array of all block hashes we request a body for.

- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.5 GetReceiptsMsg

- Request ID
- GetReceiptsRequest - array of all block hashes we request a receipt for.
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.6 AddPeer

- Peercount
- Peer Details - name, id, address, connection
- Direction - inbound/outbound
- Time - Time.Now()

A.1.7 RemovePeer

- Peercount
- Peer Details - id, address
- Direction - inbound/outbound
- Connection Duration
- Time - Time.Now()
- Disconnect Reason

A.1.8 PeerTable

- Action - Node Added/Removed
- Node ID, Address
- Timestamp_logged - time.Now()

A.1.9 GetPooledTransactionsMsg

- Request ID
- GetPooledTxRequest - array of all transaction hashes requested.
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.10 PooledTransactionsMsg

- Hashes - array of TxHashes
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.11 BlockBodiesMsg

- Request ID
- Array of blockbodies corresponding to the hashes sent in the GetBlockBodiesMsg.
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.12 BlockHeadersMsg

- Request ID
- Array of block headers corresponding to the hashes sent in the GetBlockHeadersMsg.
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.13 ReceiptsMsg

- Request ID
- Array of receipts corresponding to the hashes sent in the GetReceiptsMsg.
- Timestamp_received - msg.Time
- Timestamp_logged - time.Now()

A.1.14 NewBlockMsg

This returns an error, so never logged. Support for non-merge networks dropped.

A.1.15 NewBlockHashesMsg

This returns an error, so never logged. Support for non-merge networks dropped.

Note: NodeDataMsg and GetNodeDataMsg were included in the original loggy implementation, however, as per EIP-4938 these messages are no longer included.

A.2 Consensus Client

A.2.1 Info

- Dictionary received from RPC Call. Name, enode, IP etc of prysm client.

A.2.2 PeerTableLog

- peer ID
- Address
- Peer Added/Removed
- Time of Action - time.Now()
- Implemented in Prune() and Add() functions - status.go

A.2.3 Attestations

- peer ID
- Msg.ID
- Address
- Peer Metadata
- Time of Action - time.Now()
- Attestation

A.2.4 peerInfo

- peer ID
- Address
- Peer Metadata
- State
- Time of Action - time.Now()