KRISTIAN HORVE TJESSEM
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Simulating Swarm's storage incentive

Master's Thesis - Computer Science - June 2023

```go
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
	if len(opts) < 1 || len(opts) > 2 {
		return nil, fmt.Errorf("wrong number of options: %d", len(opts))
	}
	c = &Configuration{}
	for _, opt := range opts {
		switch v := opt.(type) {
		case gorums.NodeListOption:
			c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
			if err ≠ nil {
				return nil, err
			}
		case QuorumSpec:
			// Must be last since v may match QuorumSpec if it is interface{}
			c.qspec = v
		default:
			return nil, fmt.Errorf("unknown option type: %v", v)
		}
	}
	// return an error if the QuorumSpec interface is not empty and no implementati
	var test interface{} = struct{}{}
	if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
		return nil, fmt.Errorf("missing required QuorumSpec")
	}
	return c, nil
}
```

“Programming is a nice break from thinking.”

– Leslie Lamport

Abstract

Swarm is a decentralized storage platform consisting of several independently operated nodes. The nodes store data units called chunks. Storing chunks allows them to partake in Swarm's storage incentive. The storage incentive is a monetary incentive for people to operate nodes. This thesis investigates how the incentive is distributed based on simulation results. Analyzing the equality of the distribution and comparing the outcome of node operators with a single node and those with multiple.

A simulator of the storage incentives was designed and created. Then, simulating several configurations for 100s of thousands of rounds for the reward distribution to get an insight into how it's distributed among nodes. Then some Swarm nodes were run in a virtual machine to test the resource usage and viability of running multiple nodes.

In a network where node operators can only have one node each, the simulation shows that a network with ideal conditions has an almost equal outcome. The outcome is significantly less equal if the network allows nodes to partake in more than one neighborhood. The reward distribution in the storage depth implementation is found to be influenced by the number of nodes and storage depth. Increasing storage depth without a proportional increase in nodes leads to fragmented neighborhoods with varying levels of participation.

When one operator can run multiple nodes, my finding is that too many nodes have a diminishing return. This happens because the nodes of the operator start to compete among themselves instead of competing against others.

Since the simulator was made modular, it was, for example, easy to change stake distribution while having the same network and vice versa. This allowed for quickly testing different configurations. The results are saved to an SQLite database that allows data analysis without rerunning the simulator.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor Leander Nikolaus Jehl for his advice, comments, and guidance.

Many thanks to Vahid Heidaripour Lakhani for his insight into Swarm.

Lastly, I would be remiss in not mentioning my family, especially my parents for their support over the years.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 Background

Swarm, in summary, is decentralized storage that allows you to store files by splitting them into chunks and uploading them to a network of nodes [Swarm-team(2021)].

Swarm aims to achieve a node network that stores chunks resiliently and redundantly [Trón(2020)]. Resilience and redundancy come from having several nodes spread worldwide storing the same chunks [Trón(2020)]. If a single node fails or is under some DOS attack, then there will be another node/path in the network [Swarm-team(2021)]. This means that Swarm does not have a single point of failure [Trón(2020)].

Since Swarm is decentralized, it does not, compared to cloud storage, have a single point of failure [Trón(2020)]. Applications that use Swarm have better service continuity; if a service provider goes bankrupt, the service usually ends, but on Swarm, the application is functioning as long Swarm is operational [Trón(2020)]. Cloud storage providers have complete control over what is allowed on their platform, which the Swarm creators argue is something that can and eventually lead to censorship [Trón(2020)]. They also say that centralized control can lead to the manipulation of opinions by controlling how and what data is displayed [Trón(2020)].

Nodes that store the chunks are doing so at a cost, coming from hardware, electricity, internet access, etc. Standard costs for running computers. To not rely on altruism alone, Swarm implements two systems to reward node operators monetarily [Swarm-team(2021)]. One system tracks bandwidth usage between nodes, and the other from storing chunks uploaded to the network [Trón(2020)]. More on Swarms' inner workings in chapter 3.

Swarm has around 5000 active nodes in the past 30 days of writing, and about 2600 nodes in total have staked [1], more on node staking in chapter 3.3.1.

1.1.2 Motivation

My thesis focuses on rewards from storing chunks, how it is distributed, and the distribution's fairness. Swarm is new and still under active development; therefore, I want to investigate how rewards are split between nodes in the network. Because Swarm offers an alternative to traditional cloud storage, it needs competitive pricing to bring in users. It also needs to balance this with the payout to node operators. If the earnings from running a node do not, at the very least, cover the operational cost, I speculate that many will not want to start operating nodes. Therefore Swarm needs to have a properly functioning rewards system that allows new operators to get started.

The storage incentive in Swarm is created to encourage node operators to want to host and to limit the amount of data uploaded to Swarm by adding a cost [foundation(2022)]. As explained later in chapter 3, a game of chance is responsible for distributing rewards from chunks. A node's chance to win is directly affected by how much it has staked [Trón(2020)]. Since nodes can affect their chances, it is worth looking into how significant that effect is. If a node can entirely dominate by having a significant stake, I speculate that new operators looking to join the network will be heavily discouraged.

A node operator will presumably want to maximize their earnings and, as such, run multiple nodes. It is possible to run multiple nodes on a single computer [Maizels(2023a)]. The operational cost is presumably internet access and electricity. Since increasing the number of nodes will mainly affect hardware utilization, it can be seen as an increase in electricity cost. Multiple nodes can potentially bring in diminishing returns if they start competing with themselves. If that happens, running as many nodes as possible is not optimal.

[1] Numbers from: `swarmscan.io`

1.2 Objectives

Investigating how the storage reward is distributed among node operators. Multiple variables can affect the distribution outcome, and since Swarm is still under active development, historical data is unreliable. For example, the blog post [foundation(2022)] mentions an update to the fixed price. The price is fixed and was just updated to a new one. The post also mentions it was the last time a manual price adjustment was set and that the following update will implement a rent oracle to decide the price. Therefore I need to simulate data.

As will be explained later in chapter 3, the storage incentive has rounds. In each round, a winner is selected. One round is about 15 minutes [foundation(2022)]. Creating a simulator that can simulate it faster than real-time will give much more data to analyze.

A flexible simulator that can be configured to test different Swarm configurations. Stake is something a node can use to increase its chances of winning, more in chapter 3. A simulator that can test different stake distributions while the node network stays the same between simulations.
Simulator goals:

- Faster than real-time.
- Modular to allow for testing of different configurations.
- Generate data for analysis.

Data analysis goals:

- Investigate the rewards distribution.
- Investigate the effect of stake.
- Test different implementations of the network structure.
- Test if there are diminishing returns from running multiple nodes.
- Gather some data about nodes' resource usage.

Lastly, I want to gather resource usage by running multiple nodes in a virtual machine.

1.3 Approach and Contributions

I created a modular simulator in Golang. It allows for testing different configurations of Swarm and its parts. It can simulate 360 000 rounds in less than a second. Since 1 round is about 15 minutes, 360 000 rounds is a little over ten years. Saving those rounds to disk is the limiting factor. It can take an hour to save all rounds to disk. Saving every 100 rounds instead is much quicker, taking me only about 40 seconds to complete.

Created modules for the Simulator:

- Four network structures.
- Three stake distributions.
- One rent oracle implementation.
- One Postage contract implementation.
- Two configurable methods of saving node states.

Findings from the data:

- Stake has a significant impact.
- Storage depth is a better model than closest nodes.
- Increasing depth pushes reward distribution toward equality.
- Increasing the number of nodes in storage depth implementations, pushes the reward distribution to be more like the stake distribution.
- Data points to an optimal number of nodes to operate.
- Optimal number of nodes to maximize the reward for an operator is tied to storage depth, operated nodes, and total nodes in the network.
- It is viable to run multiple nodes.

1.4 Outline

Chapter 2 shows related work.

Chapter 3 explains the inner workings of Swarm used to develop a simulator of the storage incentive.

Chapter 4 starts by outlining the requirements for the simulator. Then the design of the simulator is presented. Then an explanation of how it is implemented. Then the approach for how the simulations were done.

Chapter 5 shows the results from running the simulations using the created simulator.

Chapter 6 explains how resource usage of a virtual machine running a Swarm node was captured. The approach for running nodes in a virtual machine. It also contains the results from the measurements.

Chapter 7 is the discussion of findings in the simulated data.

Chapter 8 contains the conclusions that are drawn from the findings.

Chapter 2

Related Work

Characteristics

The study presented in [Xu(2023)] focuses on the analysis of two decentralized storage networks, IPFS and Swarm, with the aim of gaining a deeper understanding of their characteristics and behavior. The key challenge addressed is the development of effective incentive protocols to reward network participants for their contributions.

Bandwidth incentives

In the study conducted by the authors of [Heidaripour Lakhani et al.(2022)Heidaripour Lakhani, Jehl, Hendriksen, and Estrada-Galiñanes], the focus was on simulating the bandwidth incentive in the Swarm network. To assess the fairness of the reward distribution, the authors utilized the Gini coefficient. By analyzing the Gini coefficient, they were able to identify and evaluate the fairness characteristics associated with the sharing of rewards among nodes.

Chapter 3

How Swarm works

3.1 Overview

Swarm is a system that is supposed to function as a distributed storage, with many independent nodes working together [Trón(2020)]. Its networking topology is set up using Kademlia [Trón(2020)], creating neighborhoods of nodes that store the same data [Trón(2020)]. The size of the neighborhood can be said to be the redundancy of the network; nodes in the same neighborhood synchronize the chunks they are storing with each other [Trón(2020)].

A chunk is a data unit uploaded to the network with a max size of 4KB [Trón(2020)]. If a user wants to upload a more than 4KB file, it must be split into several chunks [Trón(2020)].

To not rely on people's altruism for storing chunks, there is a payment to upload chunks to the Swarm [Trón(2020)]. Like postage stamps, a stamp-like system exists where a user buys a batch of stamps that gets ”attached” to the uploaded chunks, allowing the Swarm network to take rent from the batch to which the stamps belong [Trón(2020)]. A batch of stamps has a tracked balance, and it's from the batch balance that the rent is taken from [Trón(2020)]. When the batch balance reaches 0, nodes can move the chunks whose balance is emptied into their cache storage, where it will eventually get garbage collected [Trón(2020)].

Swarm uses smart contracts on a blockchain to enforce its storage incentive [Maizels(2023b)]. At the time of writing, Swarm uses the Gnosis chain [Maizels(2023b)], which is a sidechain to Ethereum [Gnosis(2023)].

Swarm itself does not produce any applications [Trón(2020)]. Instead, it aims to be the drive and sets itself up so that other developers can create applications that presumably use Swarm for storage [Trón(2020)].

3.2 Networking

Swarm has an overlay and underlay network [Trón(2020)]. The overlay determines who is connected to who; it is the topology of node connections [Trón(2020)]. The underlay network is how the nodes communicate with each other, and it uses TCP/IP for node communication [Trón(2020)]. This means a node will have two addresses, an overlay address and an underlay address [Trón(2020)].

3.2.1 Overlay - Kademlia

The overlay network uses Kademlia [Trón(2020)].

Kademlia is a distributed hash table (DHT) protocol designed for decentralized peer-to-peer networks [Wikipedia contributors(2023)].

A node uses its blockchain wallet address as its Kademlia address [Trón(2020)]. Each node maintains a routing table that keeps track of other nodes in the network [Trón(2020)]. The routing table is divided into multiple buckets, with each bucket responsible for a specific range of node IDs [Trón(2020)].

3.3 Smart contracts & a token

Swarm has some smart contracts that are used for its storage incentives [foundation(2022)]:

- Staking contract.
- Rent oracle.
- Postage contract.
- Redistribution contract.

To enable monetary exchange, a token called BZZ is used [foundation(2022)]. The token is paid to the contracts and received as a reward in the storage incentive system [foundation(2022)].

3.3.1 Stake & the staking contract

Staking is when a node sends some BZZ to the staking contract to be eligible to participate in the storage incentive [foundation(2022)]. It is a way for the node to put some monetary value on staying honest; if they misbehave, they lose their stake [Trón(2020)]. The current implementation does not allow the stake to be withdrawn [Raja(2023)].

3.3.2 Rent oracle

It is the rent oracles' job to decide the rent, which is set by looking at the network's redundancy, where redundancy is the number of nodes in a neighborhood [Trón(2020)]. Since the goal is a redundancy variable of 4, one can adjust the price to encourage nodes to join or leave the network [foundation(2022)].

3.3.3 Postage contract

In the postage contract, one can purchase a batch of stamps [Trón(2020)]. A batch id is received from the purchase that is then used when signing a chunk [Trón(2020)]. Parts of a stamp [Trón(2020)]:

- Chunk address
- Batch id
- signature

Other users besides the original buyer can deposit more funds to a batch after its creation [Trón(2020)].

3.3.4 Redistribution contract

For every Nth block on the blockchain, the redistribution contract randomly generates an anchor (Kademlia address) to find a neighborhood of nodes [Raja(2023)]. Nodes in that neighborhood must calculate a reserve hash from the chunks it stores [Raja(2023)]. Nodes only use chunks that are part of the neighborhood's responsibility [Raja(2023)]. They will not have the same reserve hash as other nodes in the neighborhood if they include their cache, as it may differ [Raja(2023)]. Nodes then hash the reserve hash with a random reveal nonce, overlay address, and storage depth to create a commit hash to submit to the redistribution contract [Raja(2023)]. This is called the commit phase of the contract [Raja(2023)].

After the commit phase, there is the reveal phase [Raja(2023)]. Nodes must submit their reveal nonce and storage depth to the smart contract [Raja(2023)]. They are included in the next phase only if the commit hash is correctly re-hashed [Raja(2023)]. If not, they are blocked from participating for some time [Raja(2023)].

The claim phase is the last; it starts by selecting a node to be a truth-teller [Raja(2023)]. It randomly selects from the nodes with a weight based on stake, meaning the higher the stake, the higher the chance of being selected [Raja(2023)]. From the nodes that agree with the truth-teller, a node is again selected randomly proportional to their stake to be the final winner [Raja(2023)]. The winner can then claim their winnings from the postage contract [Raja(2023)].

One iteration of these phases is called a round [foundation(2022)]. Thus every Nth block, the contract starts a new redistribution round.

This game of chance is set up as a coordination game [Raja(2023)], meaning the optimum strategy is to perform the same actions as the rest of the players [Wikipedia contributors(2022)]. To discourage nodes from misbehaving, their stake will be reduced or even entirely removed if found to break protocol [foundation(2022)].

An overview of the process created by the creators of Swarm is in figure 3.1.

3.4 Incentives

Swarm has two primary incentives, one with a focus on interactions between nodes (bandwidth), and the other is incentives for actually storing chunks [Trón(2020)].

3.4.1 bandwidth incentive

Nodes have a ledger that tracks bandwidth usage between themselves and other nodes they communicate with [Swarm-team(2021)]. Since nodes request things from each other, one can say that node A owes B if A has sent a request to B [Swarm-team(2021)]. A can then pay off its debt to B by paying them some BZZ tokens or waiting a period [Swarm-team(2021)]. The debt also gets reduced if B were to request something from A since it equals out the debt [Swarm-team(2021)]. The debt has a threshold value that, if reached, the node that is owed will stop responding until the debt is paid off or sufficient time has passed [Swarm-team(2021)].

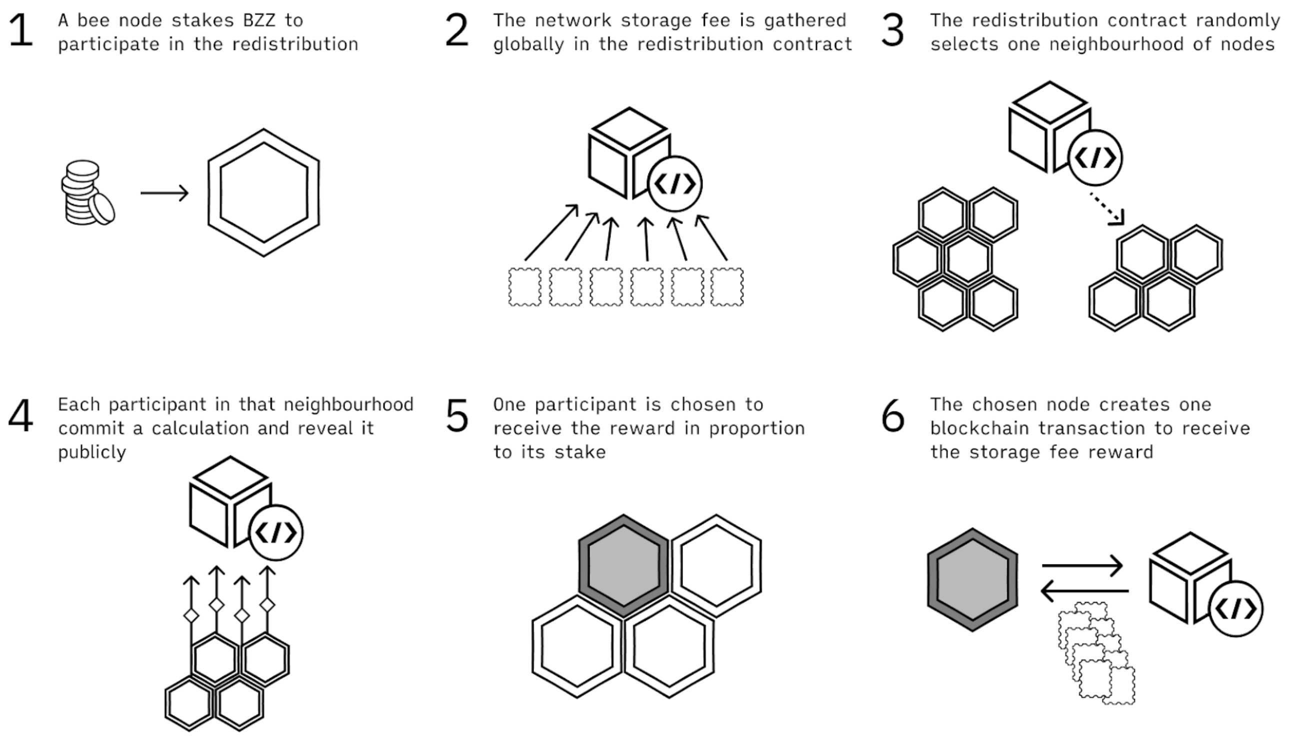

1
A bee node stakes BZZ to participate in the redistribution
2
The network storage fee is gathered globally in the redistribution contract
3
The redistribution contract randomly selects one neighbourhood of nodes
4
Each participant in that neighbourhood commit a calculation and reveal it publicly
5
One participant is chosen to receive the reward in proportion to its stake
6
The chosen node creates one blockchain transaction to receive the storage fee reward

Figure 3.1: Overview taken from [foundation(2022)]

3.4.2 Storage incentive

Storage incentive is the combination of the mentioned smart contracts in Section 3.3 to create a system that is self-regulating [Trón(2020)]:

- The postage contract controls what content (chunks) is paid for.
- A new round starts every Nth block on the blockchain.
- Rent Oracle figures out the rent for the round.
- Staking contract controls which nodes are allowed to partake in redistribution rounds.
- The redistribution contract distributes the collected rent for the round.

Chapter 4

Creating a simulator

4.1 Requirements

To gain insight into how the storage incentive is distributed among nodes, a simulator needs to model the parameters that affect how a node wins. I have gathered that three different parameters can affect a node's chance of winning. First is the stake, a value the node has complete control over. Secondly, how a neighborhood is selected; a node can not control it. And third, running multiple nodes allows a node operator to increase their overall chance of winning.

4.1.1 Stake distribution

The influence of stake on a node's chances of winning in Swarm's storage incentive presents an opportunity to explore various stake distributions. Examining different distributions of stake among nodes gives insight into the effects on reward distribution.

Equal stake

A baseline scenario that involves nodes with equal stakes, where all nodes have an equal chance of winning the reward distribution. In such a case, it can be anticipated that earnings will tend to equalize among the nodes over a sufficient number of rounds.

Power law

Power law distributions show up in economics [Gabaix(2009)]. These distributions are characterized by small entities holding a disproportionately large share of resources while the majority possess a relatively minor share. Investigating how such a stake distribution affects the reward distribution, can reveal information about the fairness in the distribution process.

4.1.2 Neighborhood selection

A neighborhood is first selected, then a winner is chosen from the nodes in the neighborhood. As such, the selection process for a neighborhood will impact a node's chance of winning.

Swarm selects a neighborhood by generating a random anchor. If the address is set to a length of four, there are $2^4 = 16$ addresses possible. The selected neighborhood is the one that the anchor resides in. The following presents one baseline approach and two methods to divide the network into neighborhoods.

Ideal/Saturated Kademlia

The ideal network is meant to be a baseline for comparison with other results. It has precisely four nodes in every neighborhood, and the chance of a neighborhood being picked is the same. The number four stems from the swarm creators wanting a redundancy variable of four [Trón(2020)]. For example, if the address length is four, and the neighborhood size (redundancy variable) is four, the network is split into four neighborhoods since $2^4/4 = 4$. Assuming all addresses have an equal probability of being the anchor for a round, it can be seen as the neighborhoods having four "tickets" to win. Since all neighborhoods have the same amount of tickets, they all have the same probability of winning, which in the example becomes $1/4$. In general, $1/h$ where h is the number of neighborhoods in the network. Or $2^l/r$ where l is the address length, and r is the redundancy variable.

The ideal network is a saturated Kademlia network. Saturated meaning nodes fill all address slots. So a network with address length 11 has $2^{11} = 2048$ nodes divided into $2048/4 = 512$ neighborhoods.

Kademlia closest nodes

The closest nodes network structure allows a node to be part of more than one neighborhood. This happens when a network is not fully saturated due to how proximity order works in Kademlia. This method stems from section 2.2.5 from [Trón(2020)]. It constructs neighborhoods by considering who is responsible for an address (chunk/anchor). The responsible ones are, at minimum, the four (redundancy variable) closest nodes to the address. It is those responsible that form a neighborhood.

This way of constructing neighborhoods makes it so a node can be part of more than one neighborhood. For example, set the address length to 3 with a redundancy of 4. If an anchor address were to be 110, then the first proximity order to check is 110. Are there four or more nodes with the address 110? If there are, a neighborhood is formed, else the proximity order decrease. The next proximity order is 11; since it is no longer a full address, it is a prefix. All nodes that have that prefix are part of the neighborhood. Again ask if that prefix has four or more nodes to form a neighborhood; if not, continue. The subsequent prefixes to examine are 1 and then 0. If it reaches the prefix of 0, it can add nodes in the address space of 0xx, which can contain up to $2^2 = 4$ nodes. If all those addresses under prefix 0 are taken, and there are not enough nodes to form a neighborhood under prefix 1, then any anchor that lands under prefix 1 includes all nodes under 0. If the anchor lands under the prefix 0, it would not include any nodes from prefix 1 since prefix 0 forms a whole neighborhood of four nodes. This gives nodes under prefix 0 a higher overall chance of winning, since they are part of two neighborhoods. Figure 4.1 visually represents the example.

This is neighborhoods, as described in *the book of Swarm*; however, it appears to have been swapped out for storage depth in current versions. On the official Discord channel for Swarm, user *ldeffenb#4199* states:

> ... Neighborhoods as it relates to Kademlia Depth seems to have gone by the wayside with the introduction of the storage depth and incentives.

This method is thus investigated to see if it performs similarly to storage depth.

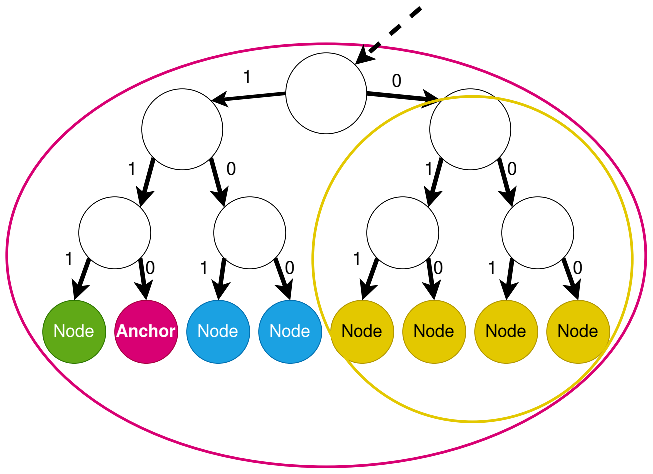


Figure 4.1: Binary tree where the nodes are leaves, no node on address 110. The four yellow nodes under the prefix 0 form a neighborhood independently. The nodes under the prefix 1 do not form a neighborhood of four. Therefore it includes the four yellow nodes creating a neighborhood of size 7.

Kademlia Storage depth

Storage depth is how Swarm, at writing, decides which neighborhood is picked for the round.

On the official Discord channel for Swarm, Discord user *ldeffenb#4199* writes:

> ... First is having the node's neighborhood selected. At the current storage radius of the swarm of 8 there are 256 (2^8) neighborhoods. ...

In a subsequent message, they explain their knowledge and experience come from reading the source code and running nodes.

So the network gets divided into neighborhoods from a storage depth variable. From the knowledge gathered from Discord, the storage depth variable increases or decreases depending on the number of chunks. Let's call the variable s. If the number of chunks in the network can not be stored with the current amount of neighborhoods, s increases, which causes the neighborhoods to split into more. Thereby needing to store less per neighborhood. It works this way since every node in a neighborhood needs to store the same chunks. Thus, when s increases, nodes will be split into new neighborhoods and be responsible for fewer chunks. The chunks they are responsible for are the ones that have the same address prefix. If the address length is three and storage depth is 2, the neighborhoods are 11, 10, 01, 00. Any chunk that has an address 11x will belong to neighborhood 11. Chunks with 01x belong to neighborhood 01 etc. Had the storage depth been 1, the neighborhoods are only 1 and 0. A visual representation in Figure 4.2.

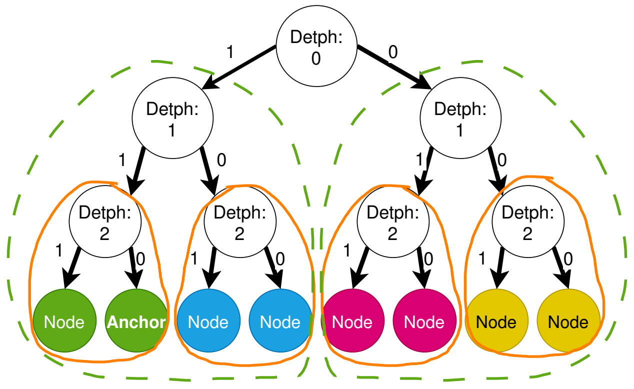


Figure 4.2: Binary tree, address length of 3. The green stapled line shows how the network is split into neighborhoods when the storage depth is 1. The solid orange line shows the network neighborhoods when storage depth is 2. At storage depth 0, all the nodes are in the same neighborhood.

4.1.3 Multiple nodes

Node operators can increase their chances of winning in each round by running multiple nodes within the network, as long as these nodes are not located in the same neighborhood. Transitioning from a single node to multiple nodes increases an operator's probability of winning.

Swarm aims to establish and maintain a decentralized network [Trón(2020)]. However, if a single node operator controls an excessive number of nodes, the network risks becoming somewhat centralized, undermining the decentralized vision. The storage incentive appears to inherently discourage node operators from accumulating an overwhelming number of nodes. When an operator controls too many nodes, the competition among their own nodes intensifies, diminishing the rewards earned by each node. If so, the outcome is a deterrent for excessive centralization tendencies, as it becomes economically inefficient for a node operator to operate many nodes.

To investigate this diminishing return, the simulator must implement a way to assign operators to nodes.

With the possibility of assigning operators to nodes, spread stake can be investigated.

Spread stake

A node operator can run one node and put their entire stake on that node, or they can spread it over multiple nodes. This distribution sets a value x that will get divided by the number of nodes an operator has. In effect, node operator one has $x/1$ stake on their only node. Operator two has two nodes, so the stake per node will be $x/2$. Operator three has three nodes, each with $x/3$ as their stake, etc.

This stake distribution ensures all operators have the same total stake in the network. It is just how it is distributed among their nodes that are different. Thus giving insight into how the chance of winning increases with the number of nodes.

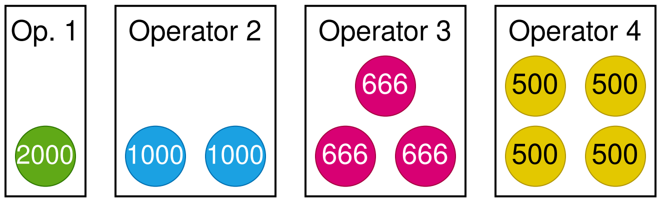


Figure 4.3: Operator 1 has one node with 2000 stake. Operator 2 has two nodes with 1000 stake each. Operator 3 has three nodes with 666 stake each, rounded down to not exceed 2000 in total. Operator 4 has four nodes with 500 stake each.

4.1.4 Analysis requirements

Storing node state

To conduct a thorough analysis and evaluation of the simulated storage reward incentive in Swarm, the simulator must incorporate a method for storing the states of individual nodes. The node state consists of the fields id, earnings, and stake. By storing node states in an external storage system, the simulator allows for post-simulation analysis of the simulation results after completion.

Gini coefficient

The Gini coefficient is a measurement tool to assess the level of equality within a distribution [world bank(2023)]. It quantifies the degree of inequality. A Gini value of zero represents perfect equality, and a Gini value of one indicates absolute inequality [world bank(2023)]. Incorporating a method for calculating the Gini coefficient provides valuable insights into the equality of reward and stake distributions.

4.2 Design

This section explains the key elements and considerations involved in creating a simulator for the storage reward incentive in Swarm.

4.2.1 Assumptions

During the design process of the simulator, some assumptions were made.

No threat actors

The simulator does not model the presence or actions of threat actors within the Swarm network. The simulator aims to simulate and analyze the storage reward incentive among nodes. This simplifying assumption allows the focus to be on the distribution among nodes.

No threat actors allow for a significant simplification of nodes. When a node wants to partake in a round, it has to calculate a commit hash from its stored chunks. With no bad actors, it can be assumed that all nodes submit proper commit hashes. Thus there is no need to run any in-depth simulation on every node.

No mid-round network change

The simulator does not incorporate any mid-round network changes during the simulation. Once a round begins, the network structure and the participating nodes remain constant until the completion of the round. This is noteworthy since a round can be up to 15 minutes or longer.

4.2.2 The main loop

A clear roadmap for the simulator is established by structuring the main loop of the simulator. This approach helps to ensure the simulator's functionality is aligned with the intended objectives.

Drawing from the details provided in Section 3.4.2, a storage incentive round can be summarized with three steps:

- Rent Oracle calculates rent.
- Redistribution contract: select a neighborhood, then pick a winner from the neighborhood.
- Winner collects the reward from the postage contract.

These steps are part of the main loop of the simulation because they make up the round of the storage incentive.

Wanting to study how the storage incentive rewards distribution evolves, the change must be stored every round. Or the state of all the nodes in the network has to be saved, depending on the wanted data. The saving of change/state has to be part of the main loop. It needs to be placed after the winner collects their reward since that is the change in the reward distribution. The saving must be done before the start of a new round.

Nodes within the network are not guaranteed to maintain 100% uptime and continuous reliability. As mentioned in the assumptions section, no simulated change occurs during a round. That is, a node won't disconnect during the reward distribution round. This means a node will always collect its reward if it wins. In actual Swarm implementation, if a node does not collect its winnings, it will be given to the next winner when they claim the reward [foundation(2022)]. Changes in the network can be simulated in-between rounds. Effectively in the main loop, this is right after saving but before the start of a new round.

In summary, the main loop of the simulator encompasses the core steps of the storage incentive round, along with two additional steps. Within the loop, the simulator will calculate rent, select the winner, reward the winner, save distribution changes, and update the network. Figure 4.4 visually represents the main loop, depicting the sequential flow of these steps.

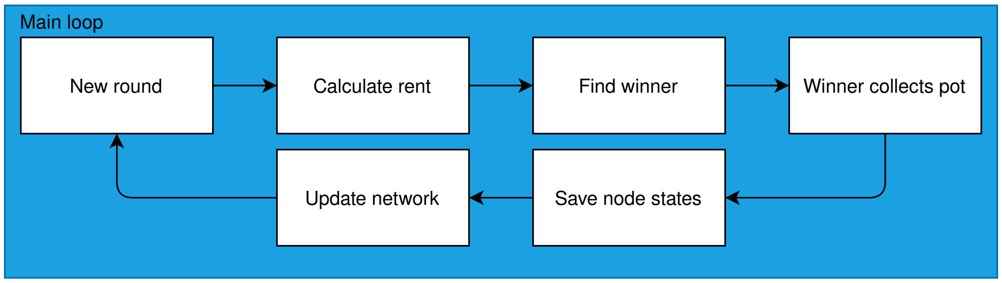


Figure 4.4: Main loop of the simulator.

4.2.3 Structure

The main loop serves as a roadmap for structuring the simulator. The starting point is a structure with a method called *MainLoop*. The main loop start by calculating rent, something done in a smart contract. Drawing from how Swarm uses smart contracts, a sort of parallel can be achieved in the code. Having the smart contracts as an interface in the code allows several interface implementations of one contract. A valuable trait that allows for testing different configurations by simply swapping to another implementation. For example, the rent oracle can have two implementations. One has a static rent every round, and the other can implement some distribution from historical data.

A structure named *simulator* has the method for the main loop of the simulator. The structure includes interfaces for the smart contracts, rent oracle, and postage as fields, allowing the main loop to call on them.

The redistribution contract is not included. Instead, selecting a winner has been shifted to the network structure. This is because selecting a winner depends heavily on the underlying network structure. First, a neighborhood has to be selected, then the winner is picked. How a neighborhood is formed is dictated by the network, as explored in Section 4.1.2. To allow for testing of different ways to select neighborhoods, a network interface is also used. Thus the main loop calls on the network interface implementation to select a winner.

A method called *setup* is also added to the structure. This is where the simulator sets the seeds for the pseudo-random number generator in Go and calls on the network to either load or create the network structure. Setting a seed is helpful if trying to replicate a result. There are two seeds, one is set before the network is created to allow for the same network structure to be made with the same seed. The other is before the main loop starts, affecting the address that wins.

Figure 4.5 visually represents the *simulator* structure.

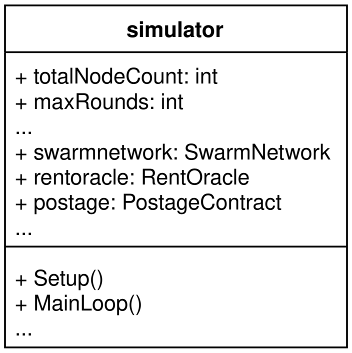


Figure 4.5: Abbriviated simulator structure.

The following explains the interfaces *SwarmNetwork*, *RentOracle*, and *Postage-Contract*. As well as the interfaces *StakeCreator* and *storer*.

SwarmNetwork

The *SwarmNetwork* interface encompasses several methods that manage the network and its nodes. The interface includes the CreateSwarmNetwork() method, which creates the network structure and generates nodes to place in the network. The UpdateNetwork() method handles changes in the network if required during the simulation. The SelectNeighbourhood() method selects a neighborhood from the network, returning a pointer to the chosen neighborhood. Similarly, the SelectWinner() method selects a winner node from the neighborhood and returns a pointer to the selected node.

The main loop never calls the SelectNeighborhood() method. Instead, it is used internally by the SelectWinner() method, which is called by the main loop.

The network implementation is responsible for creating and placing nodes in the network. Since the storage incentive is entirely independent of the bandwidth incentive, there is no need to simulate node interactions. A node can then be simplified to a data structure without any methods to handle any simulation processes. Also, with the assumption of no threat actors, there is no need to simulate nodes submitting faulty commit hashes. The simulator does not make nodes

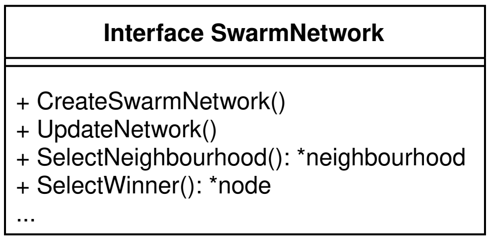


Figure 4.6: Abbreviated UML of the network interface.

submit any hashes, it assumes all nodes are operating honestly and fault free in their submissions. This allows for a simplified node structure of four fields:

- ID
- Earnings
- Stake
- Address

It is worth noting that although nodes are created in the network interface, a separate staking interface handles the stake assignment for the nodes, explained later. The staking interface is part of the structures that implement the *SwarmNetwork* interface.

RentOracle

The rent oracle is responsible for calculating rent for the round. The interface only has one method required, GetRentPrice(). It is up to the different interface implementations on how the rent is calculated.

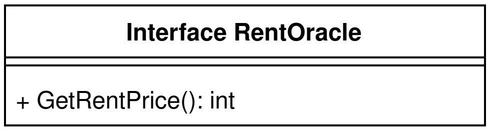


Figure 4.7: UML of the *RentOracle* interface.

PostageContract

The postage contract handles paying out the reward to the winner. As such, the interface requires the method CollectWinnings(roundPrice, *node) and GetTotalPayout() to be implemented by structures. The total payout method exists for data logging since the rent may change from round to round.

The CollectWinnings method requires the rent for the round and a pointer to the winning node. It is up to implementations of the *PostageContract* interface to choose how the reward is calculated.

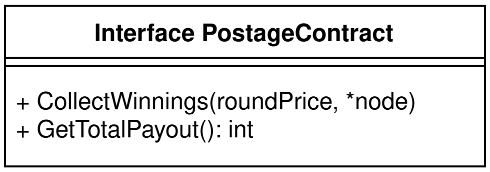

Interface PostageContract
+ CollectWinnings(roundPrice, *node)
+ GetTotalPayout(): int

Figure 4.8: UML of the *PostageContract* interface.

StakeCreator

The interface only requires the method GetStake(nodeID int). Implementations of *StakeCreator* interface decide how the stake distribution of nodes is distributed. Whether it uses the *nodeID* or not is up to the implementation.

To simplify the creation and update process of the network, I have placed the *StakeCreator* in implementations of *SwarmNetwork*. This simplifies the overall management of the network, as the stake distribution is handled within the context of the *SwarmNetwork* implementation.

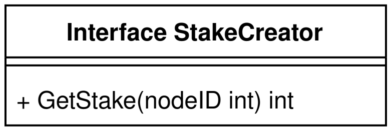


Figure 4.9: UML of the *StakeCreator* interface.

storer

The *storer* interface requires the method *save* to be implemented. Implementations of this interface are responsible for storing the data of the distributions, whether it stores all rounds, every Nth round, or just the change like a blockchain.

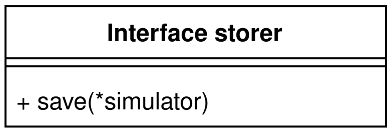


Figure 4.10: UML of the *storer* interface.

4.3 Implementation

In this section, we will delve into the implementations of the interfaces using the Go programming language (Golang).

4.3.1 Simulator

The simulator implementation has a *Setup* method that only does two things. First, it sets the seed for the pseudo-random number generator, then calls the *CreateSwarmNetwork* method that belongs to the implementation of the *SwarmNetwork* interface. The number generator is used during the network creation, so setting the seed here will always produce the same network if the same seed is given and the *SwarmNetwork* implementation is the same.

```go
func (s *simulator) Setup() {
    rand.Seed(s.SetupSeed)
    s.swarmnetwork.CreateSwarmNetwork()
}
```

In the *MainLoop* method, the seed for the pseudo-random number generator is again set. This time it is responsible for which nodes win. So if the seed is the same, the same address will be picked if the network implementation is the same. It is important to note that even if the seed is identical, it does not guarantee the selection of the same winner, as the node placement within the network may vary. The generation of anchor addresses remains predictable with the same seed. Therefore, while the same address may win, the specific node selected as the winner may differ.

After the seed has been set, the main loop starts. The main loop is looping through rounds of the storage incentive. The *simulator* structure has a configuration variable called *maxRounds*. As the name suggests, it is the number of rounds to simulate. In code, it is simply a for loop.

The for loop follows the main loop as discussed in Section 4.2.2. The main loop starts with a new round handled by the for loop. Then the rent for the round has to be calculated and stored in a variable *s.roundPrice* by calling the *GetRentPrice* method of the implemented *RentOracle* interface. After rent is calculated, a winner needs to be selected. The winner is selected by calling the *SelectWinner* method of the implemented *SwarmNetwork* interface. The winning node's pointer is stored in a variable *s.roundWinner* such that it can be passed as an argument in the next step. The winner claims their reward from the method *CollectWinnings* of the *PostageContract* interface implementation. After the reward is claimed, the implementation of *storer* interface has their method *save* called. Then the *SwarmNetwork* implementations method *UpdateNetwork* is called. The changes simulated in the network depend on the implementation.

The explanation put into code:

```go
func (s *simulator) MainLoop() {
    rand.Seed(s.simulationSeed)
    // The main loop of the simulator
    for s.round = 0; s.round < s.maxRounds; s.round++ {
        s.roundPrice = s.rentoracle.GetRentPrice()
        //select the winner
        s.roundWinner = s.swarmnetwork.SelectWinner()
        // collection
        s.postage.CollectWinnings(s.roundPrice, s.roundWinner)
        // Log changes
        s.saver.save(s)
        // Simulate change
        s.swarmnetwork.UpdateNetwork()
    }
}
```

4.3.2 SwarmNetwork

The network interface has three different implementations. These implementations are static, meaning the network does not simulate any changes. No nodes suddenly stop participating, change their stake, etc.

The difference in the implementations is how the neighborhoods are selected. But all the implementations share code for selecting a winner from a neighborhood. A winner is selected from the neighborhood by first summing the stake of the nodes in the neighborhood. Then use the *rand* library in Golang to generate a random integer in the interval $[0, S)$ where S is the summed stake of the neighborhood. Then looping through the nodes in the neighborhood and subtracting their stake from the random number. If a node brings the number to zero or below, they are selected as the winner.

```go
func (sn *FixedIdealSwarmNetwork) SelectWinner() *node {
    nbhood := sn.SelectNeighbourhood()
    weigthSum := 0
    for i := 0; i < nbhood.nodeCount; i++ {
        weigthSum += nbhood.nodes[i].stake
    }
    num := rand.Intn(weigthSum)

    for i := 0; i < nbhood.nodeCount; i++ {
        num -= nbhood.nodes[i].stake
        if num <= 0 {
            return nbhood.nodes[i]
        }
    }
    // If it gets here, something is wrong
    panic("Found no winning node")
}
```

Figure 4.11: Code snippet is taken from the ideal implementation but is identical to the other network implementations.

All current network implementations do not simulate any change in the network. The *UpdateNetwork* method is empty.

We will now explain how the network implementations are implemented based on the descriptions from Section 4.1.2. The implementations: ideal, Kademlia closest nodes, and Kademlia Storage depth.

Ideal

The ideal network means that the nodes are perfectly divided into neighborhoods of size four. The number four comes from the redundancy variable that the Swarm creators want to be four [foundation(2022)].

CreateSwarmNetwork method creates a slice that contains $N/4$ empty neighborhoods, N being the total number of nodes. Then it starts a loop that creates N new nodes. When a node is created, it gets assigned stake from the implemented *StakeCreator* interface by calling the *GetStake* method. After the node is configured, it gets placed in a random neighborhood. A neighborhood will never exceed four nodes.

The *SelectNeighbourhood* method selects a neighborhood at random. Since the chance for a neighborhood to be picked is the same for all, the implementation uses the pseudo-random generator to generate a number in the range $[0, L)$ where L is the length of the neighborhood slice.

```
func (sn *FixedIdealSwarmNetwork) SelectNeighbourhood() *neighbourhood {
    ind := rand.Intn(len(sn.neighbourhoods))
    return &sn.neighbourhoods[ind]
}
```

Figure 4.12: Neighborhood selection in the ideal network implementation.

Kademlia closest nodes

The Kademlia implementation uses a binary tree data structure for the nodes. A node's Kademlia address determines its placement within the binary tree. In this binary tree, a binary digit of zero corresponds to a right child in the tree, while a binary digit of one corresponds to a left child.

In the *CreateSwarmNetwork* method, a node is created, assigned a stake, assigned a randomly generated Kademlia address, and then placed into the binary tree. The Kademlia address is a randomly generated binary string. The binary string generator starts with an empty string and iteratively appends a one or a zero depending on a coin flip until it reaches the desired address length.

```go
func (kdst *KademSwarmTree) CreateSwarmNetwork() {
    for i := 0; i < kdst.nodeCount; i++ {
        // Create node
        n := &node{Id: uint64(i), stake: kdst.stakeDistribution.GetStake(i)}
        ...
        //Create Kademlia address.
        ...
            nAdd = randomBitString(kdst.addressLength)
        ...
        n.address = nAdd
        ...
        kdst.kademTree.InsertNode(n, n.address)
    }
}
```

Figure 4.13: Abbriviated code listing displaying the main work of the *CreateSwarmNetwork* method for Kademlia closest nodes.

The neighborhood selection process is in the method *SelectNeighbourhood*. It starts by generating an anchor (Kademlia address) using the same binary string generator that creates addresses for nodes. From this anchor, it navigates the binary tree until it reaches the address. If it arrives at the address, then add that node. It may stop early if there is no node at the given address. After it has added the leaf node or stopped due to there not being a node at the given address, it starts navigating up the tree. It checks if nodes are under the other child when it navigates upward. That is, it navigated to parent A from child B, then all the nodes under child C are appended to the neighborhood. If the neighborhood reaches a size greater than or equal to four, it returns the neighborhood. The binary tree method *FindClosestNodes* handles finding the closest four or more nodes to the anchor address.

This is because instead of finding a neighborhood, the implementation finds, at minimum, four nodes closest to the anchor and makes it the neighborhood, as explained in Section 4.1.2.

Selecting a winner from the neighborhood is done the same way as in the ideal network.

```go
func (kdst KademSwarmTree) SelectNeighbourhood() *neighbourhood {
    anch := randomBitString(kdst.addressLength)
    nodes := kdst.kademTree.FindClosestNodes(anch)
    nei := neighbourhood{nodes: nodes, nodeCount: len(nodes)}
    return &nei
}
```

Figure 4.14: Code for selecting a neighborhood in closest node implementation. Finding the closest four or more nodes is handled by the binary tree method *FindClosestNodes*.

Kademlia Storage depth

The only difference between storage depth and the closest nodes implementation is how a neighborhood is selected.

The storage depth implementation generates a round anchor and then navigates down the tree with the anchor until it reaches the storage depth. Any node that is below that tree node is allowed to partake. The navigating and returning of all Swarm nodes below the given storage depth is done by the binary tree method *navigateWithStop*.

Storage depth in this implementation is static. The value is set before the simulation starts.

```go
func (kdst *KademSwarmTreeStorageDepth) SelectNeighbourhood() *neighbourhood {
    anch := randomBitString(kdst.addressLength)
    nodes := kdst.kademTree.navigateWithStop(anch,
        kdst.storageDepth).allNodeBelowArr
    nei := neighbourhood{nodes: nodes, nodeCount: len(nodes)}
    return &nei
}
```

Figure 4.15: Code for selecting a neighborhood in storage depth implementation. Finding nodes below the storage depth is handled by the binary tree method *navigateWithStop*.

4.3.3 Stake creator

The stake creator is responsible for setting the stake in the network. This section explains the code implementation of equal stake, Power law distributed stake, and spread stake.

Equal stake

The easiest implementation. It has a variable called *amount* set before the simulation. The method *GetStake* returns the variable *amount*. This leads to all nodes having the same stake.

Power law distribution

In [Clauset et al.(2009)Clauset, Shalizi, and Newman] appendix D, formula D.4:

$$x = x_{min}(1-r)^{-1/(\alpha-1)}$$

The formula can be used to generate a power law distribution. Information from [Clauset et al.(2009)Clauset, Shalizi, and Newman]:

- x_{min} is the smallest value the formula can produce.
- α is a scaling variable with the typical range $2 < \alpha < 3$.
- r is usually a random number from a uniform distribution.

This is easily translated to code:

```
func (st PowerDistStake) GetStake(nodeID int) int {
    r := rand.Float64()
    x := float64(st.minStake) * math.Pow(1-r, -1/(st.alpha-1))
    return int(x)
}
```

Since Swarm has a minimum stake required to partake in the storage incentive, it fits in perfectly. Further logic is implemented to optionally cap a node's max stake.

Spread stake

Due to how it is implemented, the nodes must be assigned their stake in order. Meaning the code assumes that the first node belongs to the first operator. The following two nodes belong to the second operator, and the subsequent three nodes after that belong to the third operator, etc.

It achieves the spread stake by keeping track of two variables, *operator* and *nc* (node count). It divides the stake assigned to the variable *stake* by the *operator* variable. This causes operator one to have the value of *stake* as their nodes stake. Then there is some logic to control when the next operator is active. The logic counts all the nodes that get their stake. When the *nc* variable equals the *operator* variable. The *operator* variable increases by one, and *nc* is reset to zero.

```go
func (sps spreadStake) GetStake(nodeID int) int {
    st := sps.stake / *sps.operator
    *sps.nc++
    if *sps.nc == *sps.operator {
        *sps.operator++
        *sps.nc = 0
    }
    return st
}
```

Figure 4.16: Code for the spread stake distribution.

4.3.4 Rent oracle

Only one implementation has been made. It is a fixed rent oracle that returns a fixed value as rent.

```go
func (ro FixedRentOracle) GetRentPrice() int {
    return ro.fixedPrice
}
```

4.3.5 Postage contract

It is from the postage contract that a node collects its winnings. The implementation is a simple one that does not simulate chunks. It sets the winnings to be the rent of the round. Meaning a node wins what the rent oracle sets to be the rent.

A variable named *totalWithdrawn* tracks the total amount paid out.

```
func (sfp *simpleFixedPostage) CollectWinnings(roundPrice int, no *node) {
    sfp.totalWithdrawn += roundPrice
    no.Earnings += roundPrice
}
```

Figure 4.17: Code for the Postage contract *CollectWinnings* method.

4.3.6 Saving results

The node state gets stored in an SQLite database to enable data analysis after the simulation. Storing the state for every node every round is slow, so there is an option to keep every X round instead. For example, storing every 100 will require only 3600 entries instead of 360 000 when simulating 360 000 rounds. Since one entry contains all the nodes, this can grow rather large if storing all rounds.

Data saving was moved to another thread with Go's concurrency feature. This was initially done to avoid blocking the simulation thread. However, this does not achieve much since one has to wait for it to finish writing the data anyways.

Figure 4.18 shows the schema design.

Storing the state of the nodes in this way enables the calculation of the Gini coefficient in SQL. Modifying the SQL in [Medvedev(2019)] gives:

```
SELECT  1-2 * sum((earnings * (rownum-1) +
cast(earnings as float)/2 )) / count(*) / sum(earnings)
AS gini
FROM
(
    SELECT nodeID, earnings, row_number() OVER (
        ORDER BY earnings DESC
    ) rownum
    FROM nround WHERE roundID=?
)
```

The Gini coefficient of a round can be calculated from a SQL query by giving a round id. If the calculation is slow, the database might have a missing index.

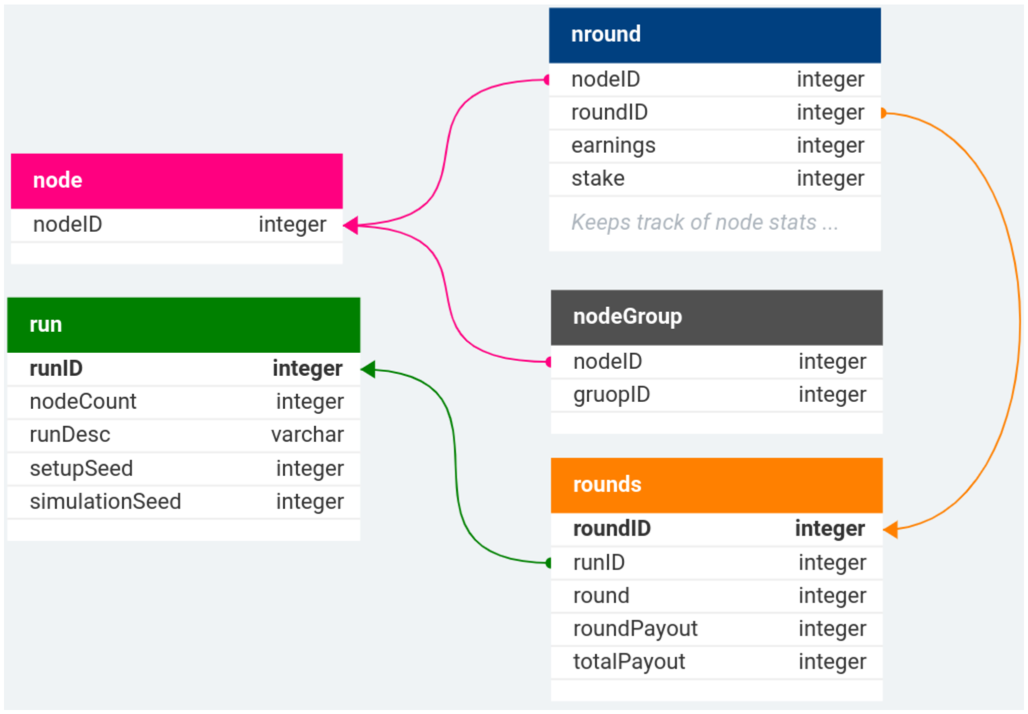


Figure 4.18: SQLite schema.

The index is intended to be added after the simulation to speed up the data entry process during the simulation.

```
CREATE INDEX idx_roundID_nround ON nround(roundID)
CREATE INDEX idx_round_runID_roundID on rounds(round,runID,roundID)
```

4.4 Approach

A file named *main.go* is used to run a simulation. At the top portion of the file, one can configure the number of nodes, Kademlia address length, rounds to simulate, database name, description of the run, set up seed, and simulation seed. These values are used to initialize structures and are collected in one place for a quick and easy overview. Example of a configuration in Figure 4.19.

The main function is defined following the configuration setup, where the necessary structures are initialized. Within the *main* function, structures have to be

```go
package main
import(
...
)
const NODECOUNT = 2048
const ADDRESSLENGTH = 128

//With 15 minutes per round, 350666 rounds is around 10 years
const ROUNDS = 350000
const DBNAME = "results.db"
const DESCRIPTION = "Simulation of 2048 nodes," +
    "static Kademlia network Storage depth, 128 bit address - Equal stake."

var SETUPSEED int64 = 123123
var SIMSEED int64 = time.Now().Unix()
...
```

Figure 4.19: Top portion of *main.go* where some configurations are set. *SETUPSEED* is set, generating a set network structure. *SIMSEED* is set to be random, so winners are random in every simulation.

initialized. That is, one must choose the interface implementations for the simulation. After the structures are set, the *simulator* structures *Setup* method is called, and subsequently, the *MainLoop* method. A continued example of Figure 4.19 is the *main* function shown in Figure 4.20.

```go
...
func main() {
    saver := saveFullStateSql{
        everyXRound: new(int),
    }
    *saver.everyXRound = 100
    saver.init()

    stake := EqualStake{
        amount: 199,
    }
    swnet := &KademSwarmTreeStorageDepth{
        addressLength:     ADDRESSLENGTH,
        nodeCount:         NODECOUNT,
        stakeDistribution: stake,
        kademTree:         bintree{root: &binNode{prefix: ""}},
        fullySaturate:     false,
        storageDepth:      8,
        addressBook:       make(map[uint64]*node),
        kademAddress:      make(map[string]*node),
        nodes:             make([]*node, 0, NODECOUNT),
    }

    s := &simulator{
        totalNodeCount: NODECOUNT,
        swarmnetwork:   swnet,
        rentoracle:     &FixedRentOracle{fixedPrice: 1},
        postage:        &simpleFixedPostage{},
        saver:          saver,
        round:          0,
        maxRounds:      ROUNDS,
        SetupSeed:      SETUPSEED,
        simulationSeed: SIMSEED,
    }
    s.Setup()
    s.MainLoop()
    saver.close()
}
```

Figure 4.20: *main* function in *main.go*. Configured to use Kademlia storage depth, save every 100th round to SQLite, with an equal stake. Rent oracle is 1, such that a node's earnings are the number of times they have won.

Chapter 5

Simulation results

5.1 Distribution of rewards

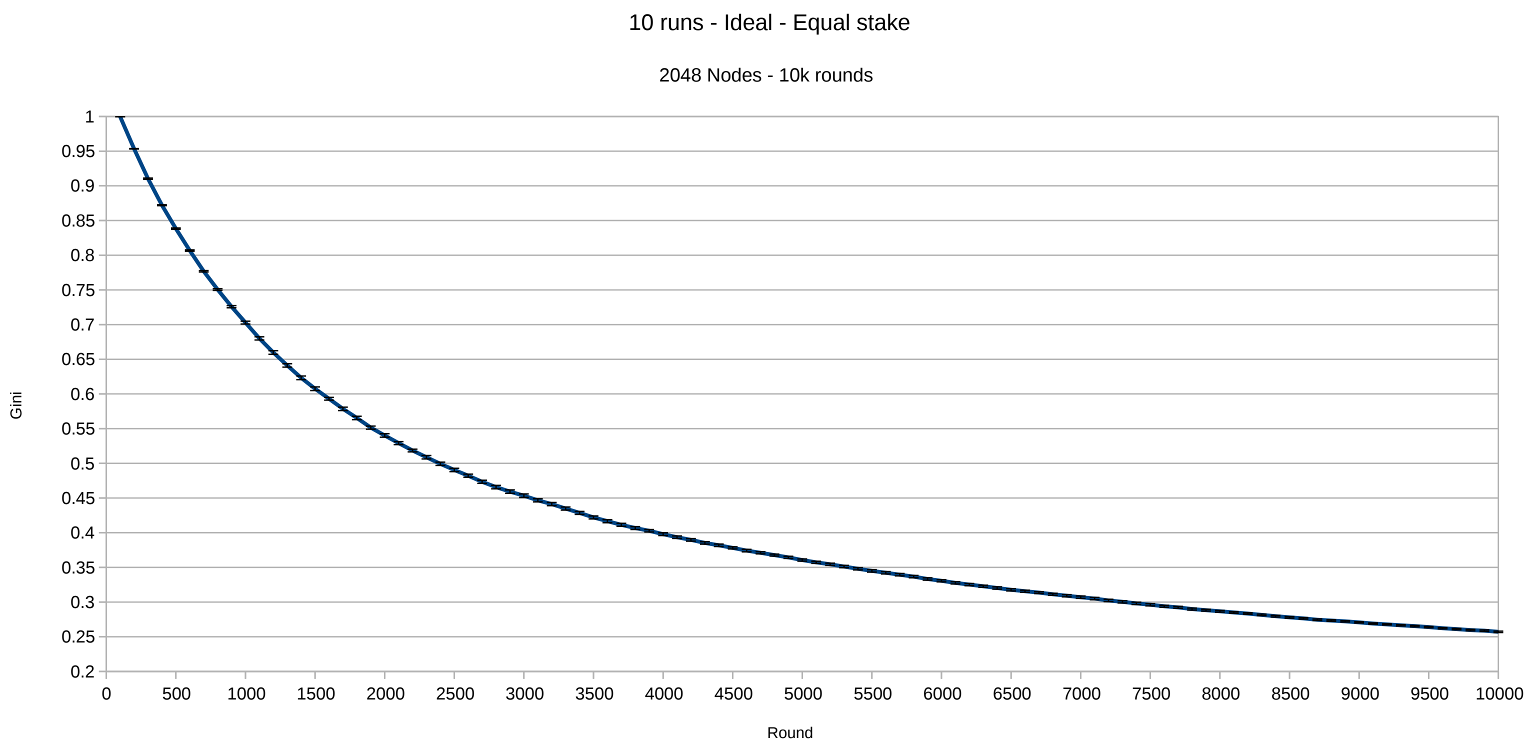
5.1.1 Equal stake

Y-axis is the Gini coefficient. It visualizes the equality of the reward distribution. X-axis is the round number.

The equal stake distribution has a Gini coefficient of zero because all nodes have the same stake.

Figure 5.1 showcases two plots obtained from the simulation of 10 runs of the ideal network implementation. Each run encompasses 350,000 storage incentive rounds, involving a total of 2048 nodes. The placement of nodes within the network and the seed for the winner selection are varied across the runs to ensure diverse outcomes. The Gini coefficient, calculated based on the mean of times won, measures the reward distribution. The top plot represents the first 10,000 rounds, while the bottom plot captures the entirety of the 350,000 rounds. It should be noted that the top plot includes barely visible error bars, which are consistently present but omitted in subsequent similar figures for the sake of clarity and readability.

The simulated ideal network with equal stake distribution confirmed the initial speculation that the rewards would become more evenly distributed among the nodes when the number of rounds increased.



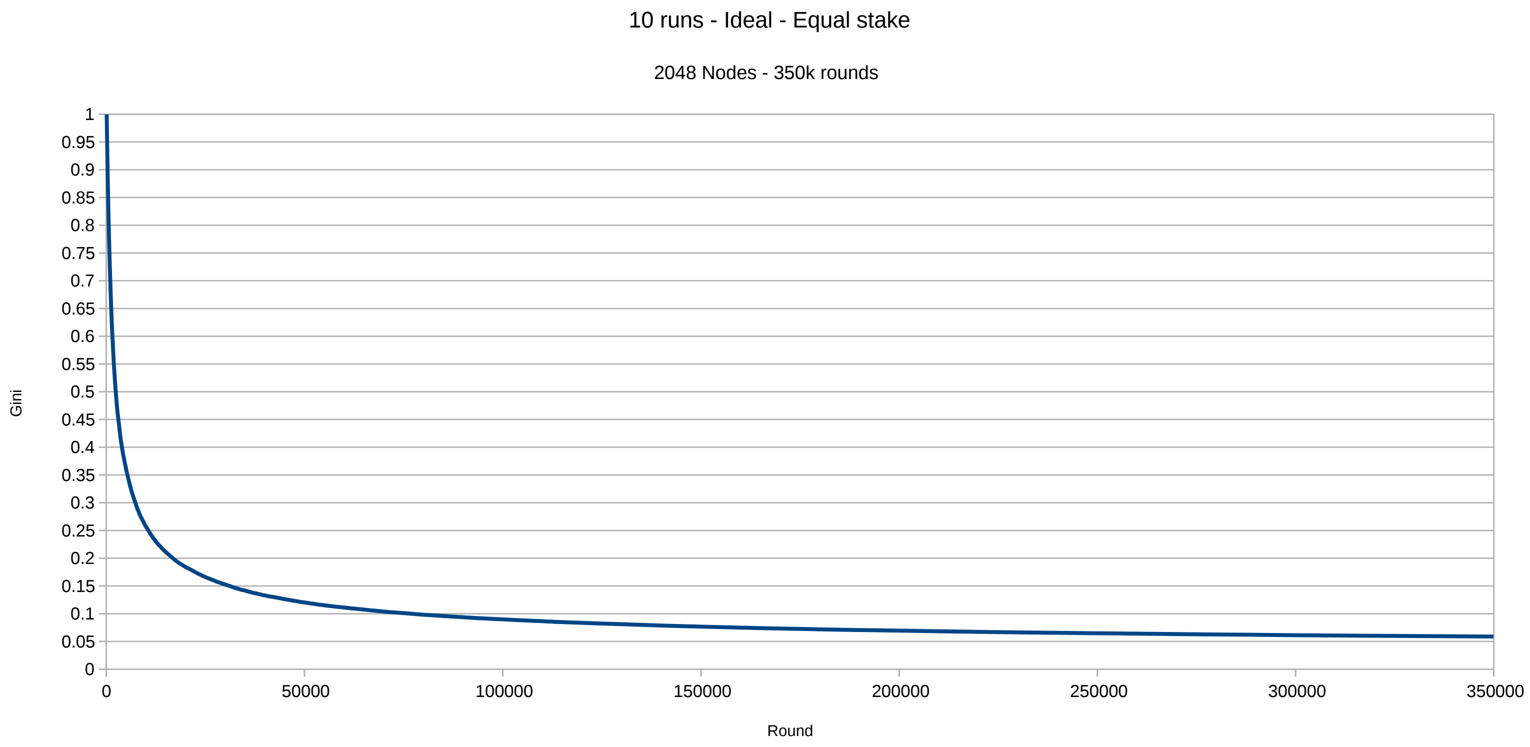


Figure 5.1: Ideal network - Equal stake.

Figure 5.2 presents the plot depicting the mean Gini results of 10 runs conducted using the saturated Kademlia implementation for 350,000 storage incentive rounds. The placement of nodes and the seed for winner selection varies between runs. The simulation confirms that the performance of the saturated Kademlia network aligns closely with the ideal network, as anticipated.

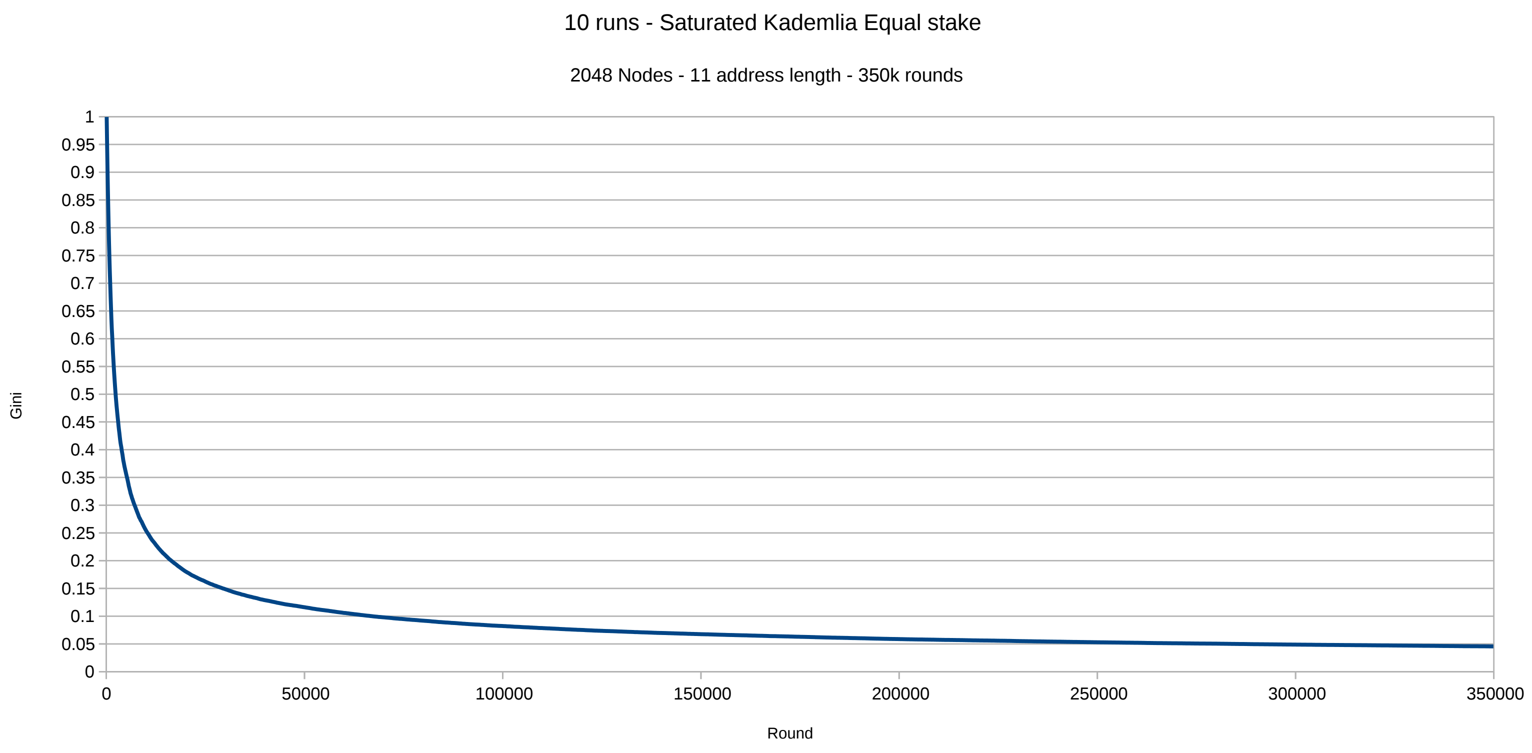


Figure 5.2: Staurated Kademlia - 11 address length. Equal stake.

A table will be provided instead of an extensive display of graphs to present the distribution outcomes concisely.

Table 5.1 contains the results of Figure 5.1 and 5.2. As well as the results from other network structures. The only difference is the network structure used. The rest of the settings are the same for all results. All runs differ in node placement and random seed for winner selection.

Table 5.1 presents the results obtained from various network structures, including those depicted in Figure 5.1 and Figure 5.2. The primary difference among these results lies in the network structure, while the remaining settings are consistent across all simulations. Notably, each run has different node placements and a random seed for winner selection, contributing to the variability in the outcomes.

Table 5.1 reveals that the closest nodes implementation falls short of achieving the same level of equality as a saturated network. A saturated network is the closest node implementation with a smaller address space, and every address is taken by a node in the network.

The findings also present in the table demonstrate that as storage depth increases, there is a corresponding increase in the inequality of the reward distribution.

Network Structure	Mean Gini	SD	SEM
Ideal	0.059	0.0009	0.0003
Saturated	0.046	0.0006	0.0002
Closest nodes	0.2669	0.0010	0.0003
Storage depth 8	0.195	0.0010	0.0003
Storage depth 16	0.347	0.0008	0.0003
Storage depth 32	0.360	0.009	0.004

Table 5.1: Round 350k. 2048 nodes. Equal stake distribution.

5.1.2 Power stake

Simulations with 2048 nodes

The results presented in Figure 5.3 illustrate the distribution of rewards in an ideal network with a Power distributed stake. The stake is distributed over 2048 nodes. Simulation has run for 350 thousand rounds. The stake distribution has a Gini coefficient of 0.48. The stake distribution remains static throughout the simulation, as no changes in stake distribution over time were simulated.

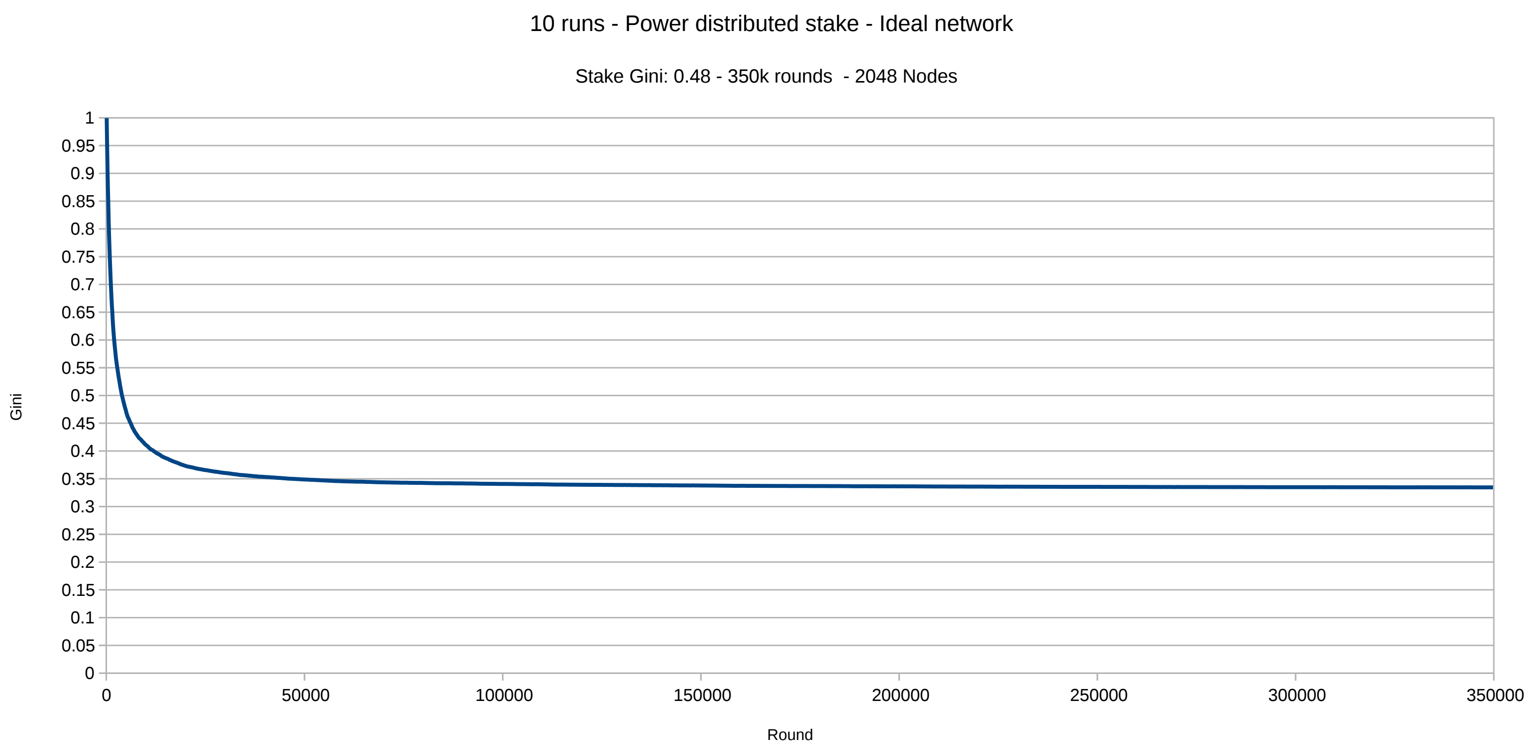


Figure 5.3: Ideal network - Power stake distribution.

Table 5.2 contains the results for the different network structures. Each structure was simulated for 10 runs, each run ended at round 350 thousand. Stake distribution remains static throughout the simulation.

The ideal network reveals that stake significantly impacts the reward distribution. Going from a value of about 6% to about 33% is a significant increase. Notably, the rewards distribution is distributed with higher equality than the stake distribution.

The closest node implementation has rewards distributed more closely to the stake distribution. Only about a 3-4% difference in how rewards are distributed compared to the stake distribution.

Storage depth shows a similar trend to the others at a depth of 8. Increasing the depth contradicts the finding in the equal stake distribution. An increase in depth gives a slightly better equality, but only to a certain depth.

Network Strucutre	Mean Gini at round 350k	SD	SEM	Stake Gini
Ideal network	0.334	0.001	0.0011	0.484
Closest nodes	0.448	0.001	0.0008	0.484
Storage depth 8	0.439	0.001	0.0008	0.484
Storage depth 16	0.366	0.001	0.0006	0.484
Storage depth 32	0.366	0.001	0.0010	0.484

Table 5.2: Power distributed stake. 2048 nodes, 10 runs.

Increasing the number of nodes

The results in Table 5.3 correspond to the closest nodes implementation with a power law distributed stake, simulated for 350 thousand rounds. The mean values are derived from 10 runs. The rewards distribution is not far from how the stake is distributed. Increasing the number of nodes in the network has minimal to no impact on the reward distribution.

Network Strucutre	Number of nodes	Mean Gini at round 350k	SD	SEM	Stake Gini
Closest nodes	4096	0.454	0.0058	0.0018	0.518
Closest nodes	8192	0.457	0.0042	0.0013	0.506
Closest nodes	16384	0.466	0.0029	0.0009	0.521

Table 5.3: Closest nodes implementation, power distributed stake, 350 thousand rounds.

The findings presented in Table 5.4 pertain to the storage depth implementation with a power law distributed stake, where the storage depth is set to 8. The mean values are derived from 10 runs with 350 thousand rounds each run. The results reveal that increasing the number of nodes in the network but keeping the same depth slightly pushes the reward distribution closer to how the stake is distributed.

Number of nodes	Mean Gini at round 350k	SD	SEM	Stake Gini
4096	0.472	0.005	0.0015	0.518
8192	0.490	0.004	0.0014	0.506
16384	0.518	0.001	0.0004	0.521

Table 5.4: Storage depth 8, power distributed stake, 350 thousand rounds.

The results obtained from Table 5.5 indicate that increasing the storage depth in the closest nodes implementation leads to a shift towards a more equal reward distribution than the results from Table 5.4. It again shows a very small trend of reward distribution moving towards the stake distribution when increasing the number of nodes.

Number of nodes	Mean Gini at round 350k	SD	SEM	Stake Gini
4096	0.379	0.005	0.001	0.518
8192	0.382	0.002	0.001	0.506
16384	0.395	0.003	0.001	0.521
32768	0.415	0.001	0.0003	0.518
65536	0.454	0.001	0.0003	0.520

Table 5.5: Storage depth 16, power distributed stake, 350 thousand rounds.

The results from Table 5.6 are from a storage depth of 16. It displays the same trends as Table 5.5. The reward distribution continues to move towards a more equitable state as the storage depth increases.

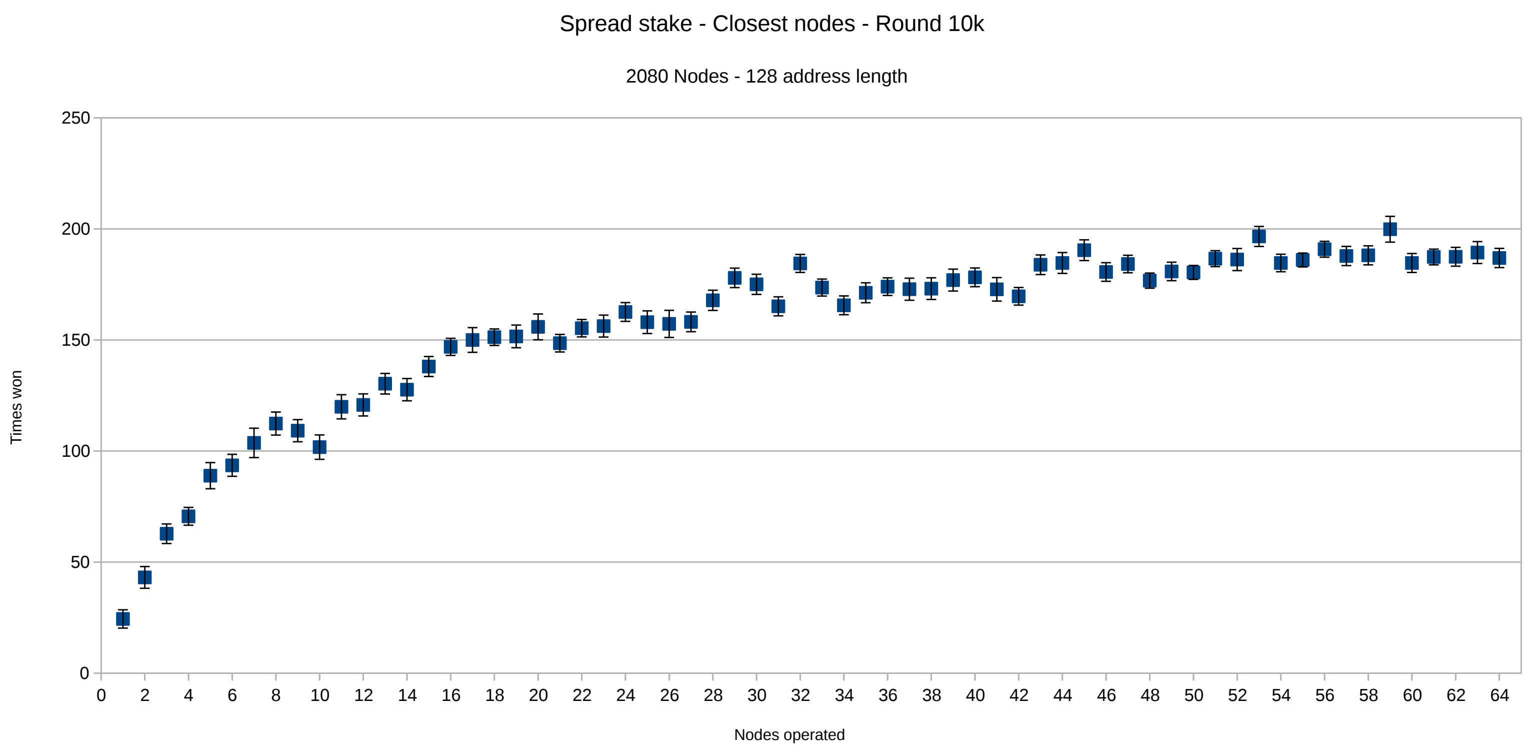
Number of nodes	Mean Gini at round 350k	SD	SEM	Stake Gini
4096	0.374	0.004	0.0011	0.518
8192	0.383	0.002	0.0008	0.506
16384	0.392	0.002	0.0007	0.521
32768	0.408	0.002	0.0006	0.518
65536	0.439	0.001	0.0003	0.520

Table 5.6: Storage depth 32, power distributed stake, 350 thousand rounds.

5.1.3 Spread stake

Results are from spread stake distribution. The stake is set to 50k. In the results with 2080 nodes, the last operator has 64 nodes. $\frac{64(64+1)}{2} = 2080$

The findings presented in Figure 5.4 demonstrate a diminishing return when running multiple nodes within the network. This observation aligns with the initial speculation that nodes belonging to the same operator begin competing for the same reward.



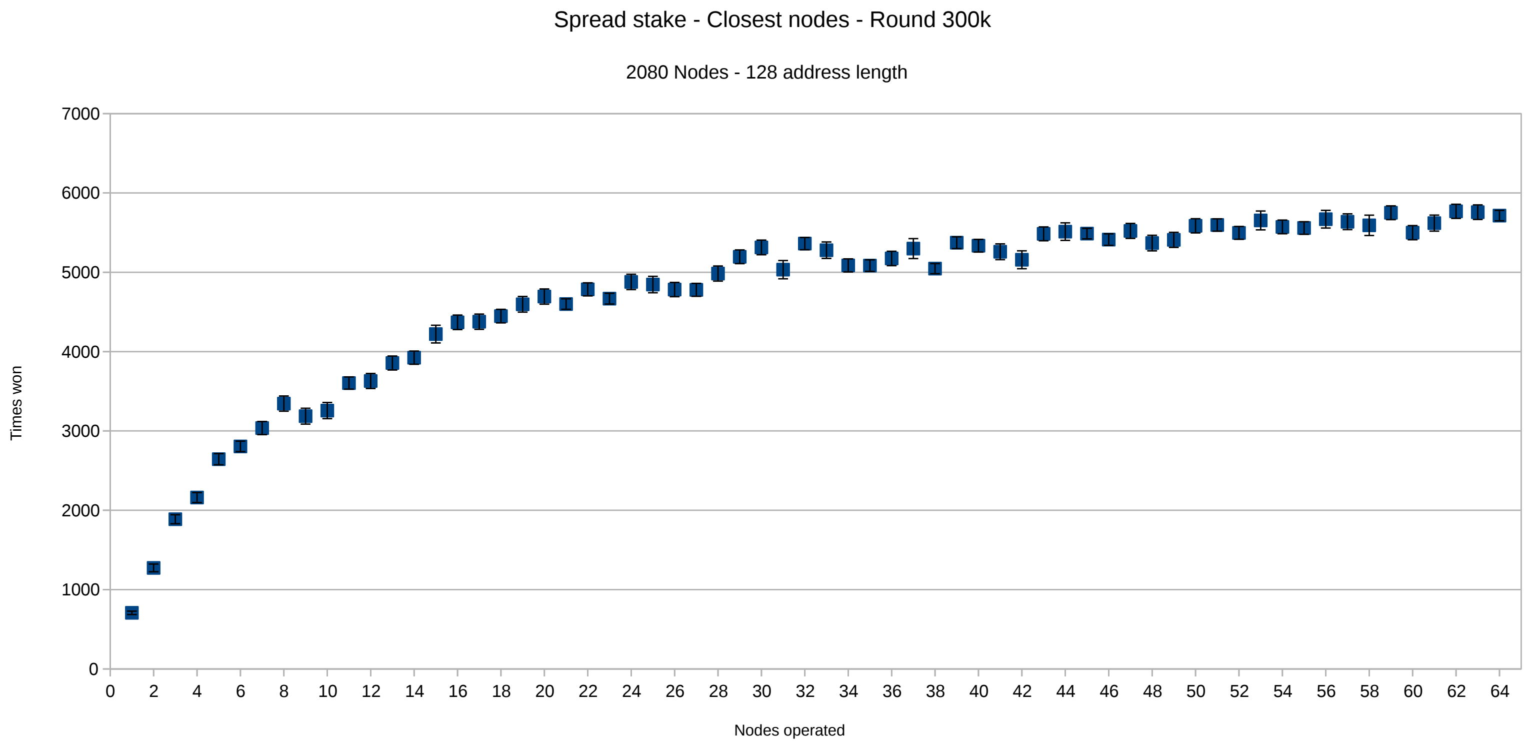


Figure 5.4: Closest nodes - Spread stake distribution.

The results from Figure 5.5 are from the storage depth implementation with a depth of 8. It shows the same diminishing return as for closest nodes implementation.

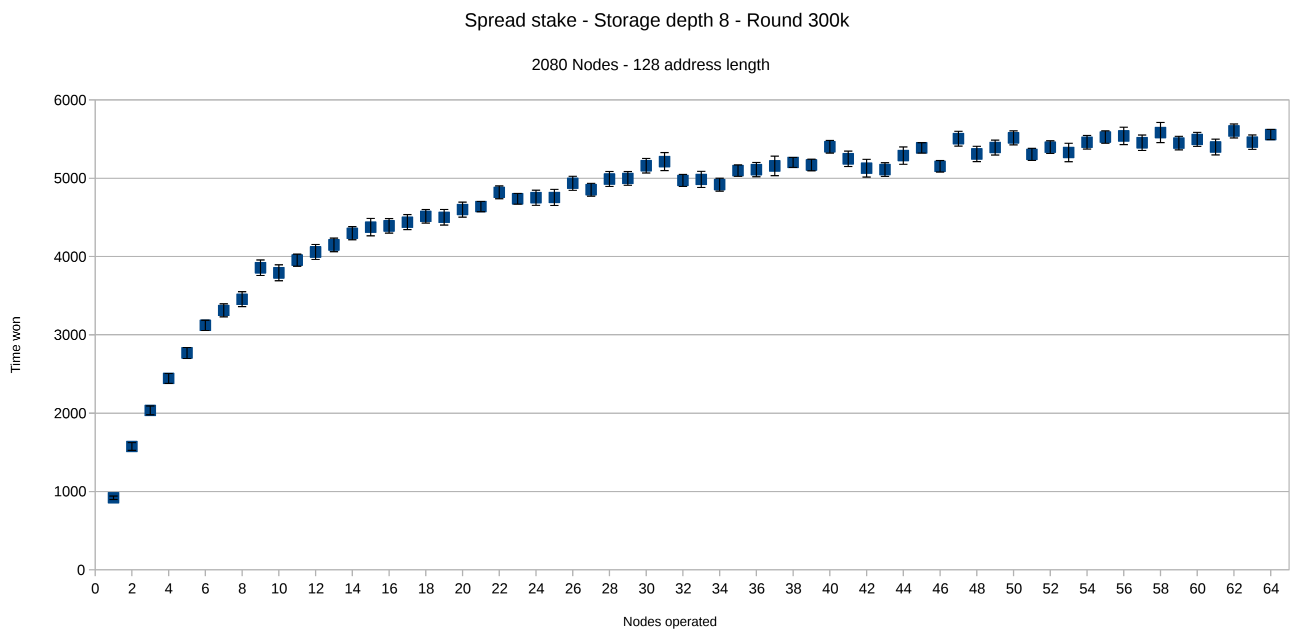


Figure 5.5: Storage depth 8 - Spread stake distribution.

Figure 5.6 is a storage depth implementation with a depth of 16. The line has become more linear, meaning there is a significant increase in winnings from running more nodes. A higher depth increases the number of neighborhoods a node can be part of, thus decreasing the chance of an operator having nodes that compete in the same neighborhood.

The results depicted in Figure 5.6 represent a storage depth implementation with a depth of 16. The plotted line is linear, indicating a notable increase in winnings associated with running more nodes. This outcome is expected as a higher storage depth spreads nodes to more neighborhoods, reducing the likelihood of an operator's nodes competing within the same neighborhood. Storage depth 32 showed the exact same.

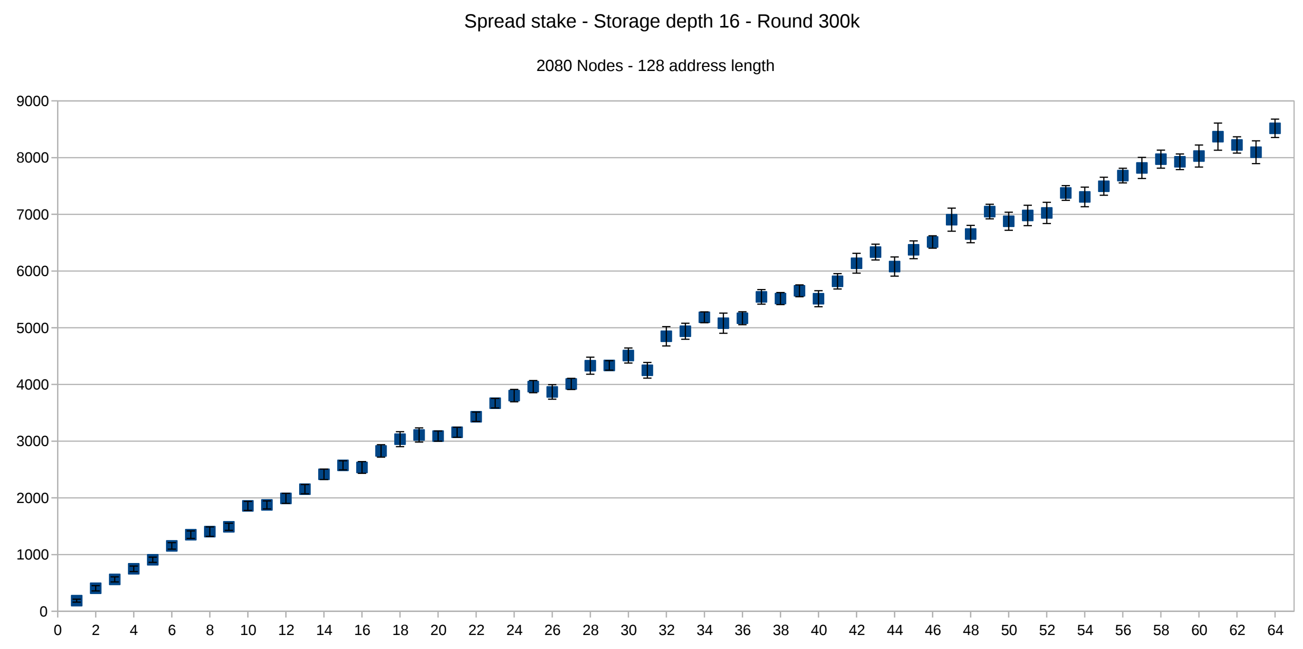


Figure 5.6: Storage depth 16 - Spread stake distribution.

Chapter 6

Resource usage approach

6.1 Virtual machine & node

A virtual machine is used to run some Swarm nodes.

Host machine specs:

- OS: Garuda Linux x86_64
- Kernel: 6.3.5-zen1-1-zen
- CPU: AMD Ryzen 9 3950X
- Drive: 1TB m.2 ssd

The VM was set up using *VirtManager*[1].

VM specs:

- OS: Debian 11
- 4 VCPUs. Not manually pinned.
- RAM: 16GB, reports 15.6GB
- Disk space: 320 GB.

A node in Swarm is called a bee [Maizels(2023a)].

[1]`https://virt-manager.org/`

6.2 Measure

I created a Python script based on the functions from *VirtManager* [2]. It is programmed in Python and released under GPL-2.0 license. It has the advantage of pulling the data from the KVM rather than running on the actual VM, avoiding the polling process being part of the data. The script pulled data from the running VM every second. Data the script collected is:

- CPU usage
- ~~RAM usage~~*
- Network usage
- Drive usage.

*Ram usage was not captured properly due to not having memory ballooning configured.

6.3 Approach

One full node was set up using the guide Swarms official guide[3]. The node was run for 24 hours, and the Virtual machine's resources were monitored.

The process for setting up a node:

1. Install Bee service.
2. Configure the Bee.
3. Start the Bee.
4. Fund node with XDAI token.

Before setting up the second nodes, an update was released to the bee nodes. It had some changes in the pull-sync protocol. As such, I performed the update and redid the 24-hour measurement.

After one node was measured for 24 hours, the subsequent measurement was of two nodes, then three, etc. Max number of nodes measured was 6.

[2] `https://github.com/virt-manager/virt-manager`

[3] `https://docs.ethswarm.org/docs/bee/installation/quick-start/`

Setting up the second, third, etc. nodes was done following the instructions. When starting a bee node, one can enter a different path for a configuration file that the node use. In the configuration file, one configures the network port it uses, places on the drive it stores its chunks, and keeps its blockchain secret key.

6.4 Results

Figure 6.1 shows the CPU usage of 1 node running version 1.15 and one running 1.16 over the span of 24 hours.

Figure 6.2 and 6.3 are the VMs network activity and drive usage with one Bee node.

To avoid flooding the rest of the thesis with graphs, I only include full stats for 1 node and 6 nodes with version 1.16.

Figure 6.4 has the CPU usage of the VM with 6 nodes. Figure 6.5 has network usage of the VM with 6 nodes. And Figure 6.6 is the drive usage of the VM with 6 nodes.

The RAM usage was not properly captured. I only have some notes, and I can say that running 6 nodes uses about 3-4 GB of RAM. Staring a node requires more memory, it slowly depletes until it settles for a value. Swarm officially recommends 8 GB[4].

[4]From: https://docs.ethswarm.org/docs/bee/installation/install

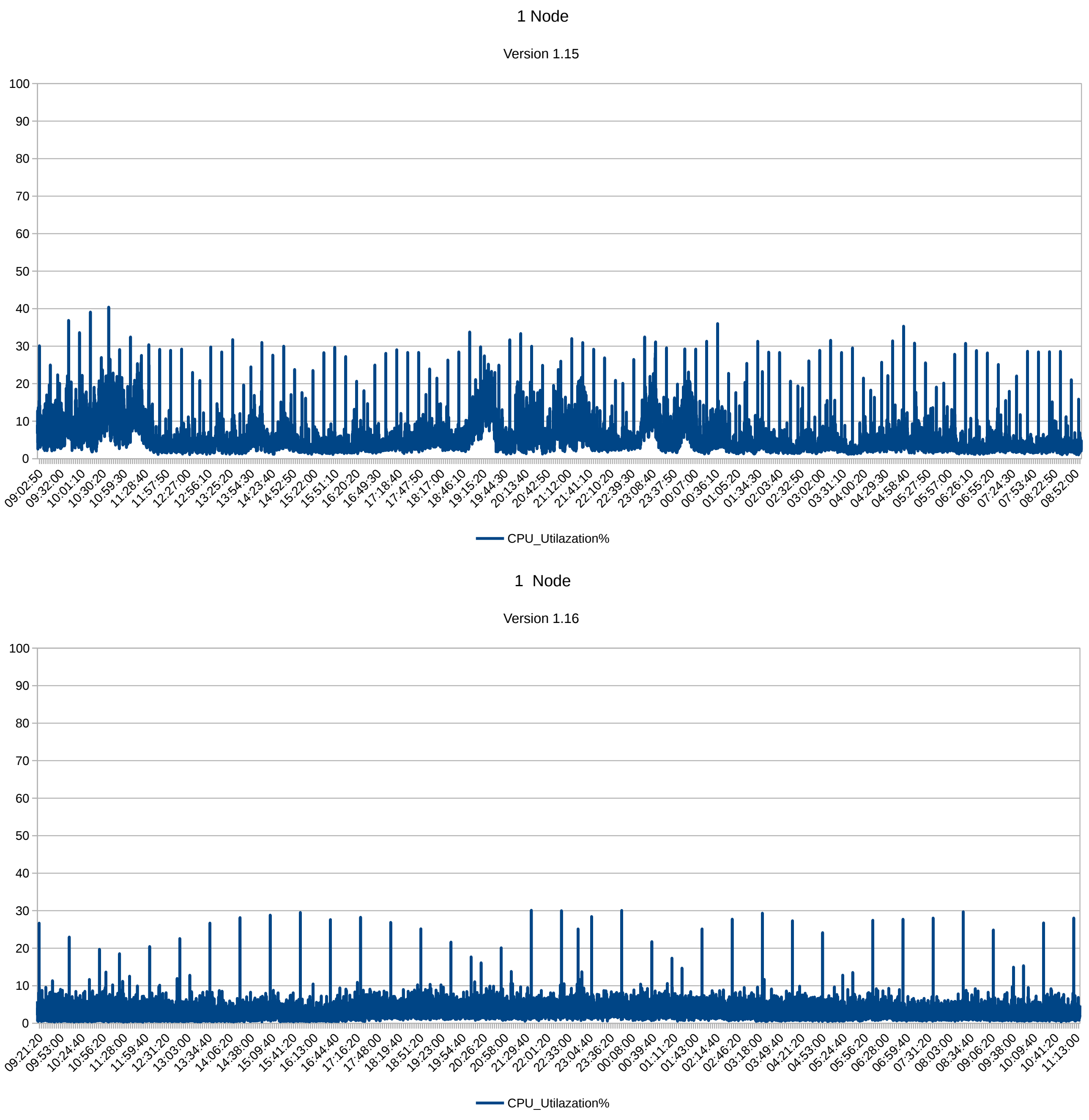
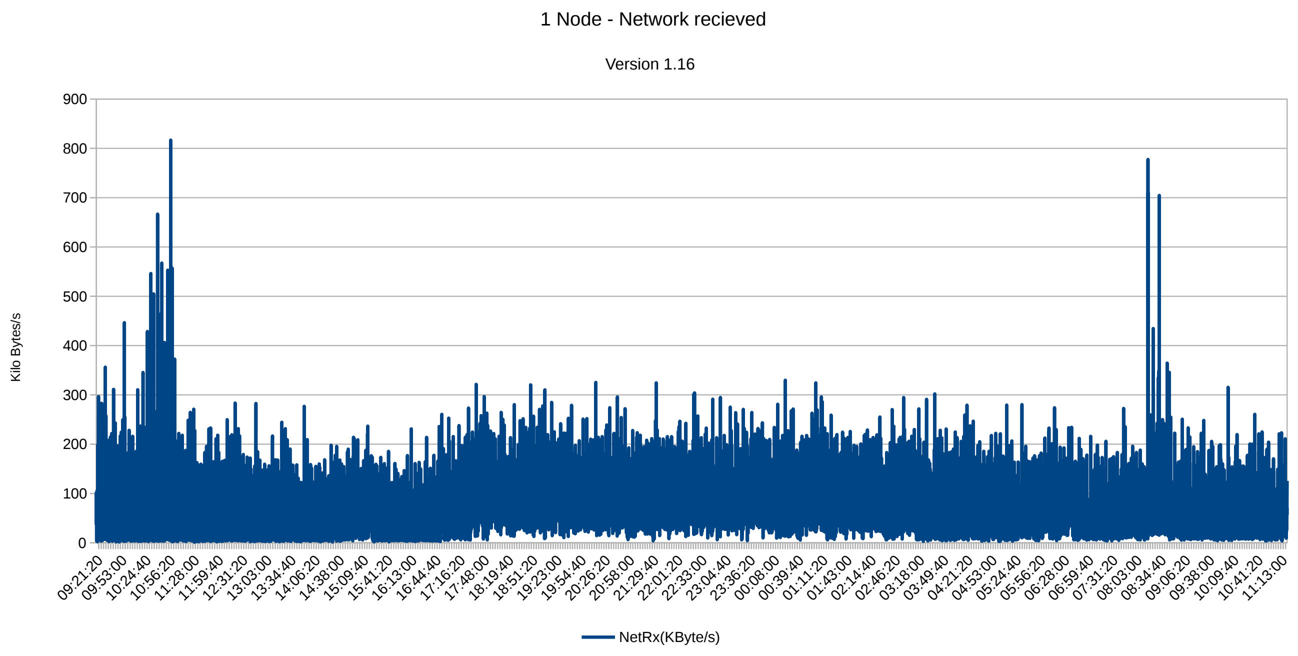


Figure 6.1: CPU utilization of 1 node. The top plot is Bee version 1.15, and the bottom plot is Bee version 1.16



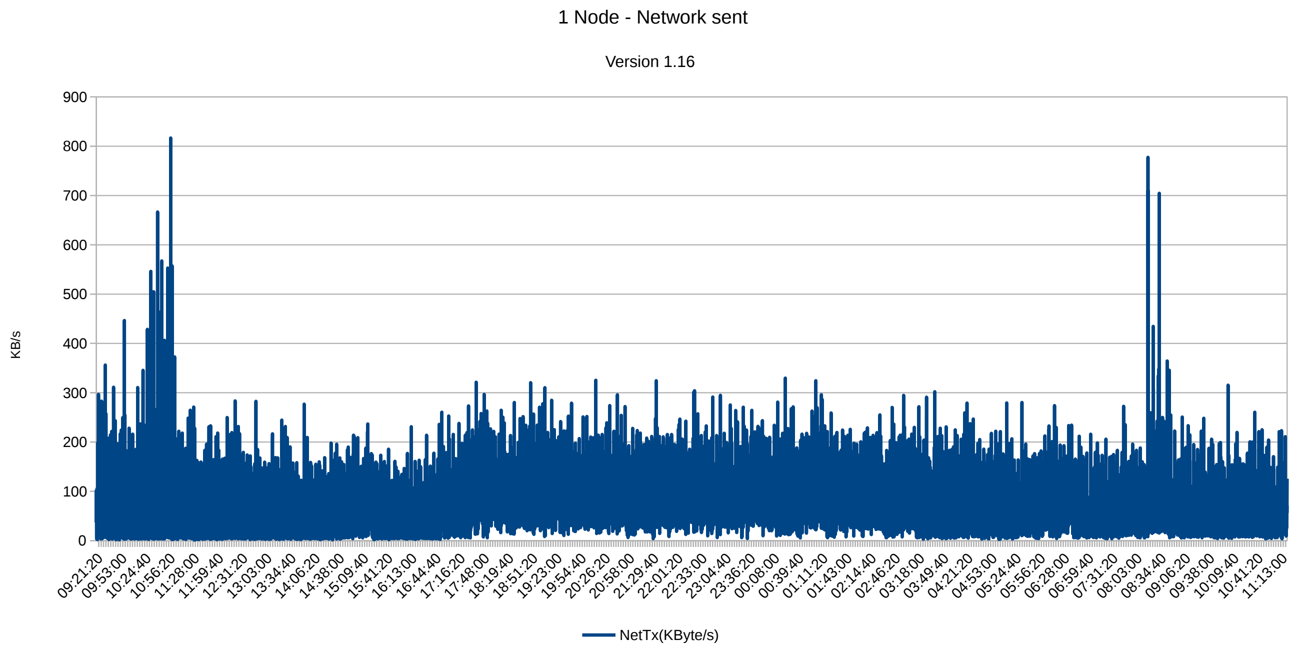
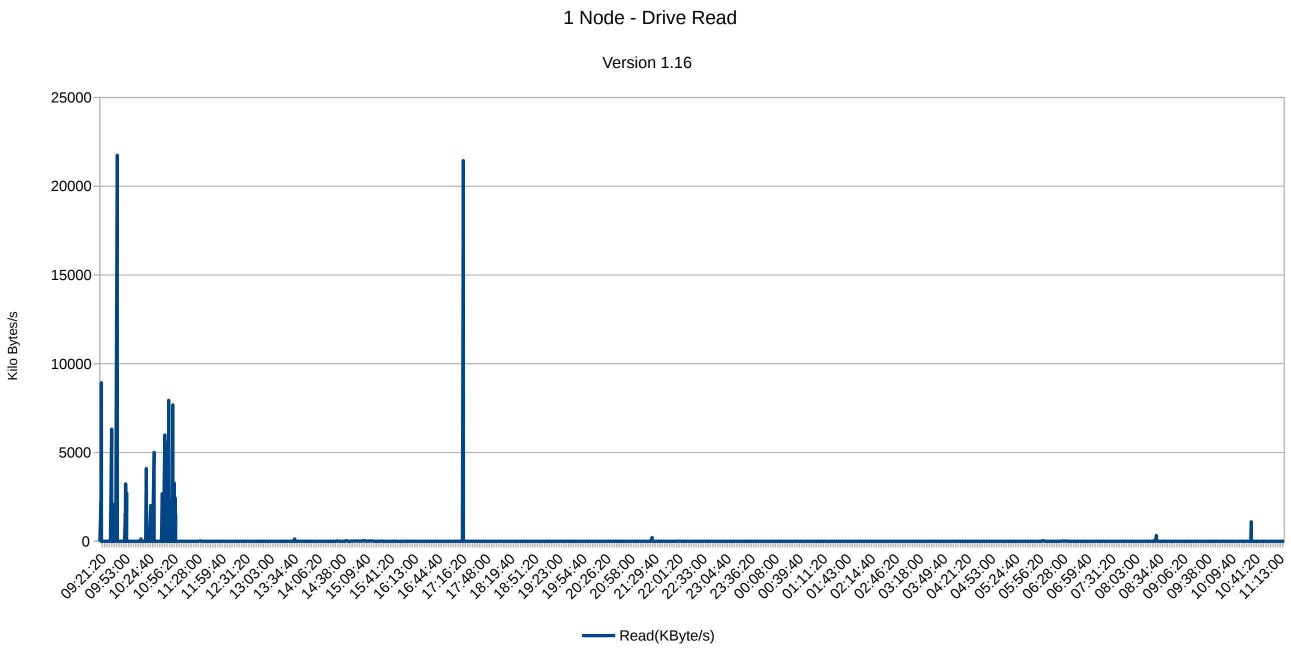


Figure 6.2: Network usage. The top plot is KB/s received. The bottom plot is KB/s sent.



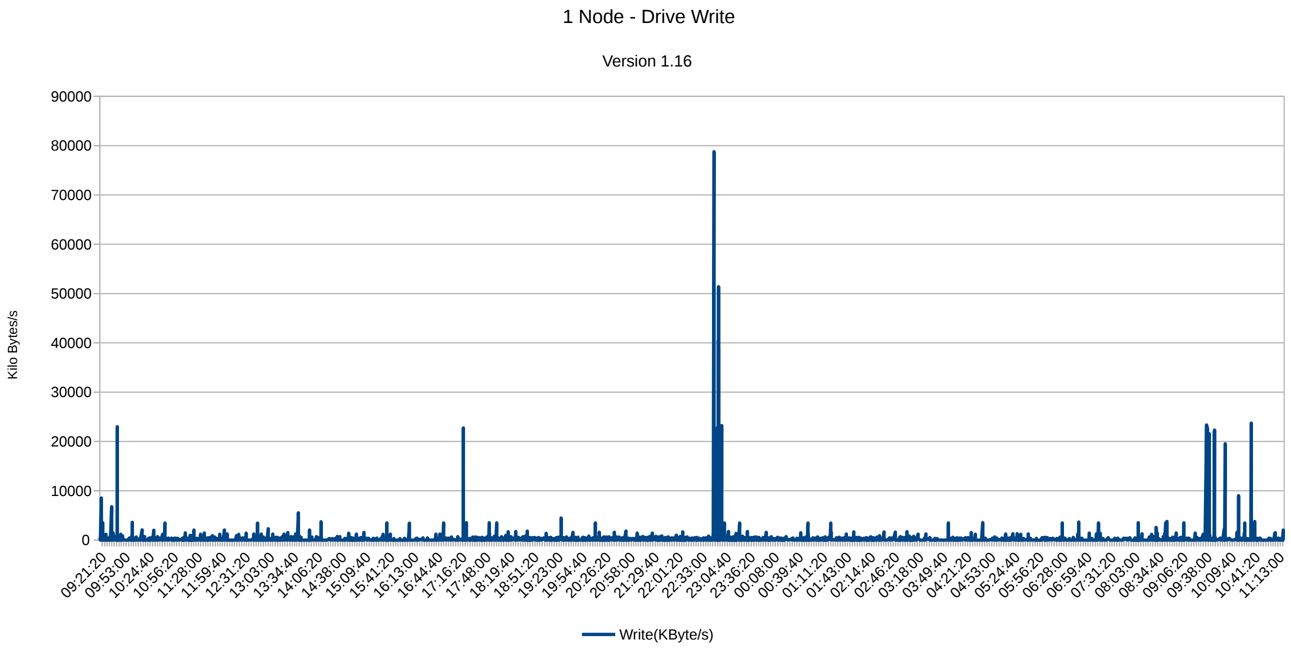


Figure 6.3: Drive usage. The top plot is KB/s read. The bottom plot is KB/s write.

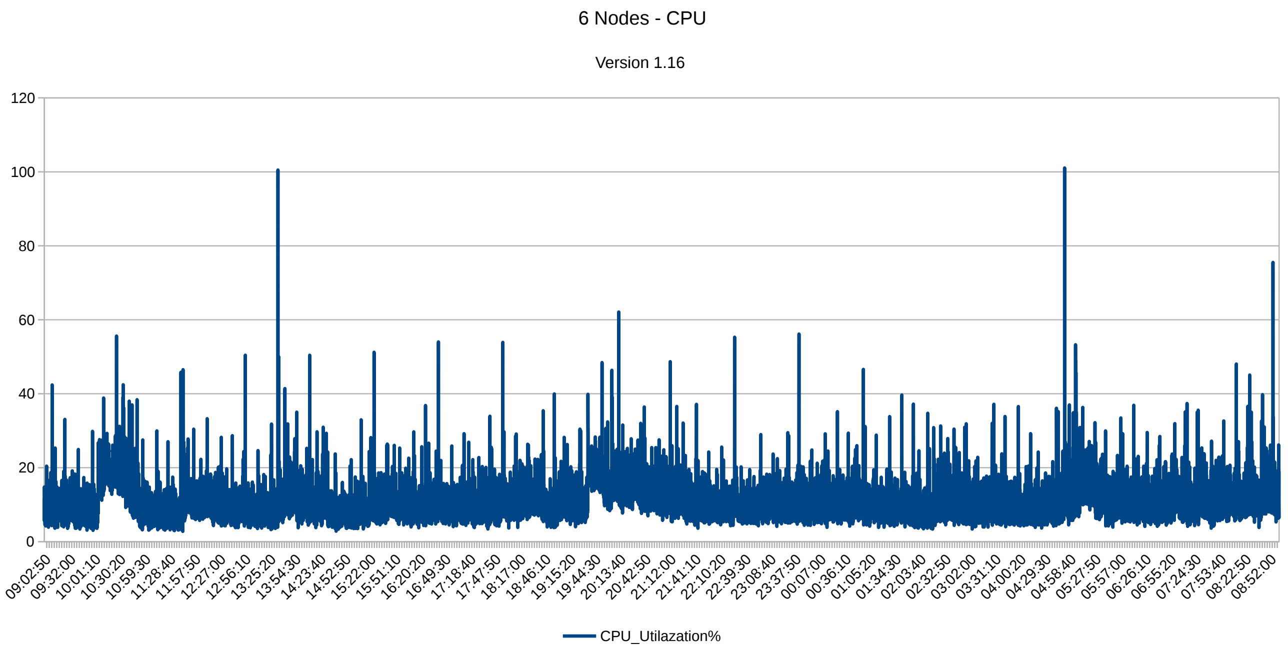
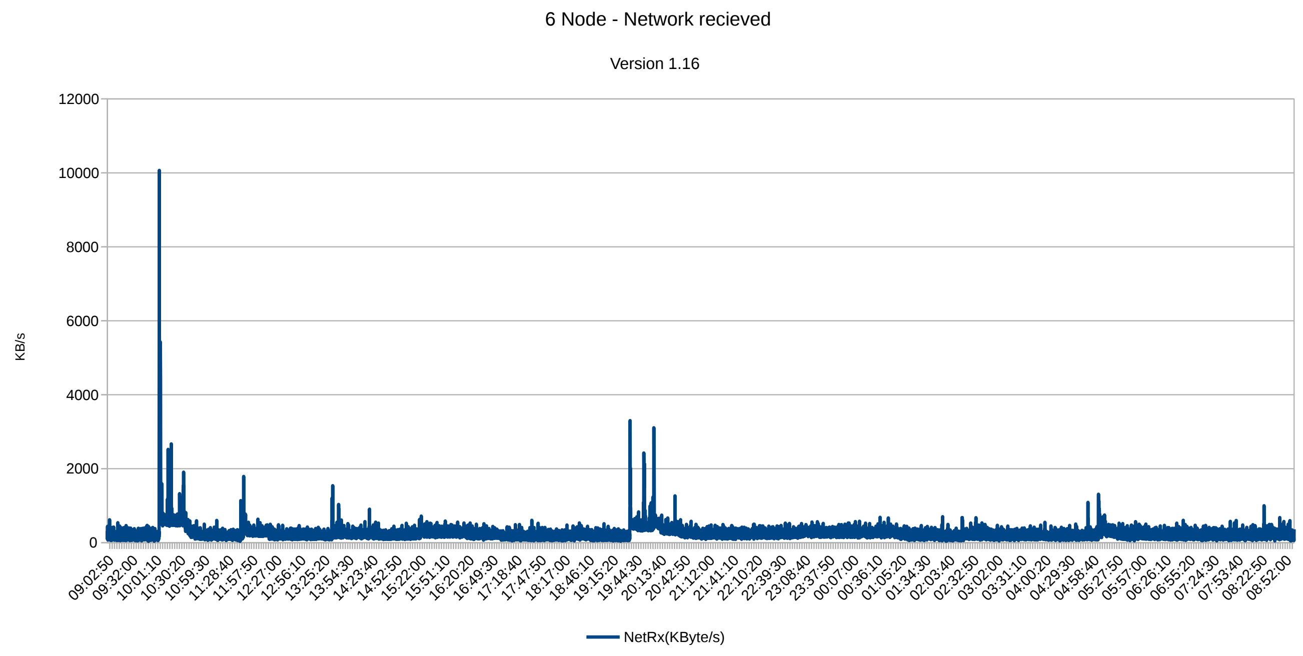


Figure 6.4: CPU utilization of 6 nodes.



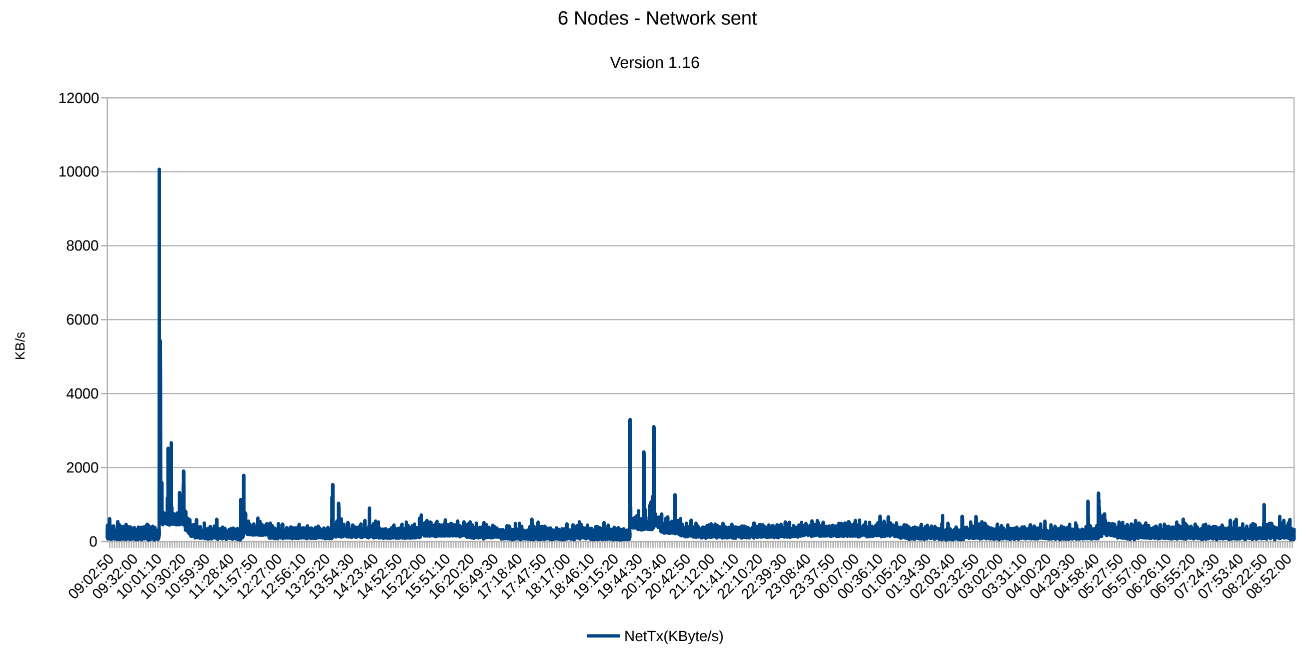
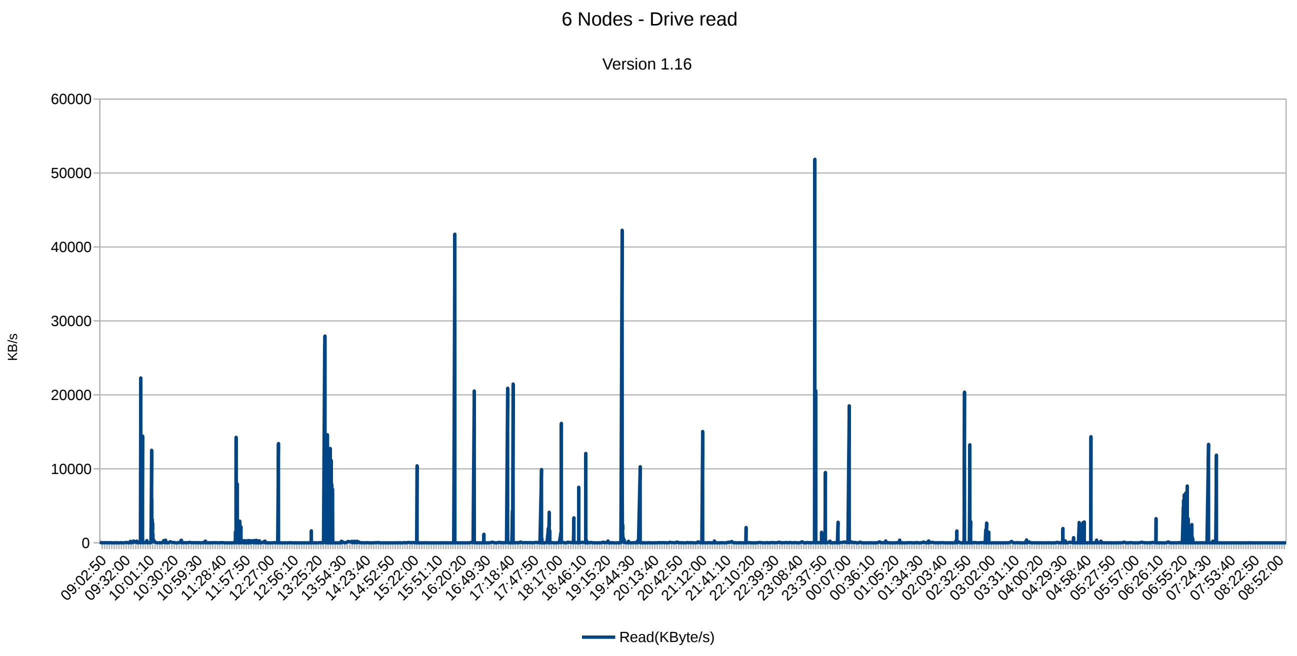


Figure 6.5: Network usage 6 nodes. The top plot is KB/s received. The bottom plot is KB/s sent.



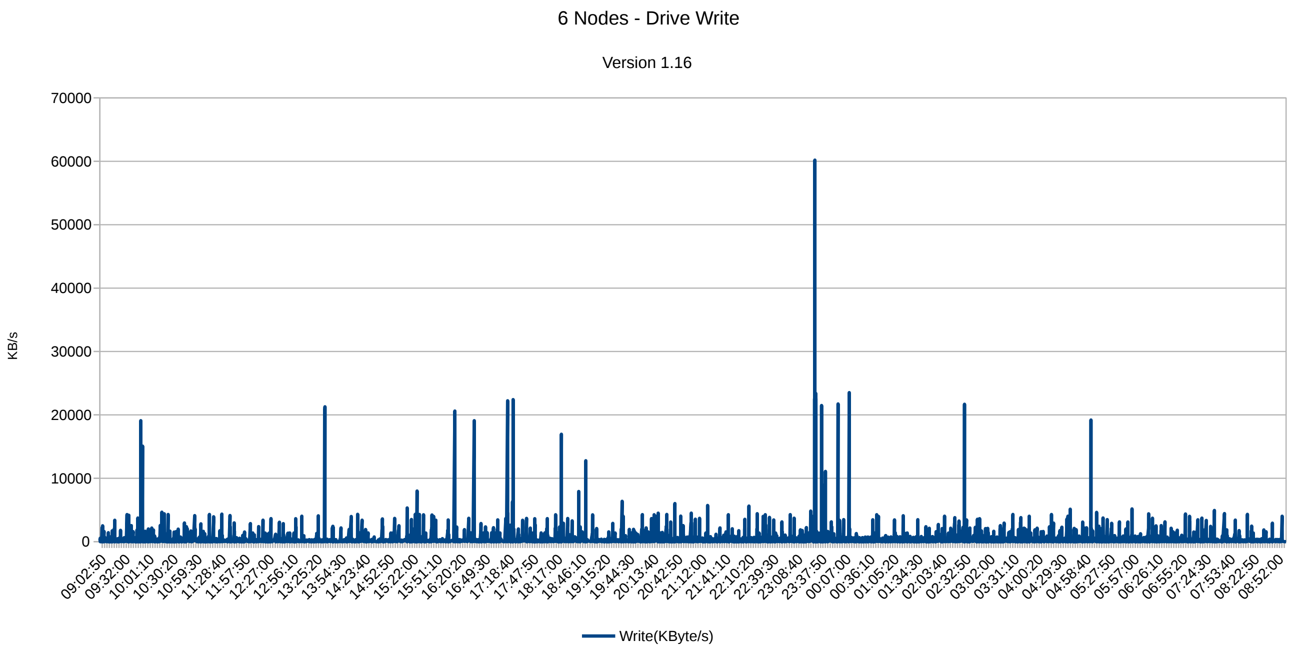


Figure 6.6: Drive usage 6 nodes. The top plot is KB/s read. The bottom plot is KB/s write.

Chapter 7

Discussion

In cases where the stake is distributed equally among the nodes, the ideal and saturated Kademlia networks produce similar outcomes. This observation aligns with the initial expectations and suggests that the network structure plays a significant role in determining the distribution of rewards.

The closest nodes implementation with equal stake does not achieve the same level of equality as the ideal network. It becomes apparent that the closest nodes approach, even with equal stake allocation, falls short of attaining the same degree of equality observed in the ideal network. Suggesting that nodes participating in multiple neighborhoods have a distinct advantage over those not. The presence of nodes in multiple neighborhoods grants them a higher probability of winning rewards, leading to an inherent imbalance in the distribution. Consequently, this finding underscores the significance of node participation in multiple neighborhoods as a factor influencing reward distribution.

The reward distribution in storage depth appears tied to the number of nodes in the network. When the storage depth is increased without a corresponding increase in the number of nodes, the network tends to be fragmented into neighborhoods with varying levels of participation. These neighborhoods exhibit significant disparities, with some encompassing 4 to 7 or more, while others consist of only one or two nodes. All neighborhoods have an equal chance of being selected. However, the number of nodes within a node's neighborhood also influences the probability of a node winning a reward.

Examining the power distributed stake highlights the impact of stake on the reward distribution within the network.

The power-distributed stake reveals that the stake has a significant influence on the reward distribution. As intended by the Swarm developers, the reward distribution seems to lean towards how stake is distributed. It did show that the ideal network and a very high storage depth compared to the number of nodes yield similar results.

The depth and the number of nodes influence the reward distribution of the storage depth implementation. If the number of nodes increases, the reward distribution moves towards the stake distribution. And if the depth increases, the reward distribution moves slightly toward equality.

Spread stake appears to clearly indicate a diminishing return when running multiple nodes. As storage depth increased, it became more advantageous to run more nodes. This makes sense because there are more neighborhoods that one node can be part of. Decreasing the chance of two or more of an operator's nodes being in the same neighborhood.

The distribution of rewards in Swarm with storage depth is a function of depth, stake distribution, and the number of nodes. From an operator's perspective, their reward is a function of depth, stake distribution, operated nodes, and total nodes in the network.

Storage depth is increased when they need more storage space, this causes the same number of nodes to have a better equality. This is an incentive to start more nodes because when the distribution is more equal, more individual nodes are winning. As the number of nodes increases but the depth remains the same, there is a lower chance of winning, which is an incentive for discouraging new nodes. This may play a part in keeping Swarm decentralized, as running a vast number of nodes does not seem to be a financial benefit.

In the closest nodes implementation, the equalizing force is only the number of nodes in the network. The main driving force for operators to start new nodes with this implementation was an increase in the monetary reward. This thesis has not made any connection to monetary value, as Swarm is still actively changing. Instead, the distribution comes from how many times a node wins.

One of the mentioned goals of Swarm is they want the storage incentive to influence the number of operated nodes in the network. Which, from this thesis, appears to be fulfilled with storage depth.

The simulator can be extended to simulate operator behavior. Apply some resource usage and cost to running a node. Then have some threshold values for when an operator starts new nodes, stops running nodes, etc. This does need to have a more robust storage depth and chunk simulation. The current implementation only has a static variable set before the simulator is run. The smart contracts were also simplified to test how the distribution evolves over time rather than actual financial gain. This is presumably closer to how Swarm is operated in reality.

Running multiple nodes in the Swarm network is a feasible option, as evidenced by results from the measurement of successfully operating six nodes. There was the miss-configuration of memory ballooning that causes script to not give accurate utilization of RAM usage

Chapter 8

Conclusions

The analysis of different implementations and stake distributions in the Swarm network provides valuable insights into the dynamics of reward distribution. The observations demonstrate that the network structure significantly influences the distribution of rewards.

Increasing the storage depth without an increase in nodes leads to fragmented neighborhoods with varying levels of participation, impacting the probability of winning rewards. The power-distributed stake highlights the strong influence of stake on the reward distribution, with the reward distribution leaning towards how stake is distributed.

The reward distribution in the storage depth implementation relies on the depth and the total number of nodes in the network. Increasing nodes push the rewards distribution closer to the stake distribution, and increasing depth leads to a slight movement towards equality.

Running multiple nodes exhibits diminishing returns, as higher storage depth decreases the likelihood of multiple nodes from the same operator being in the same neighborhood.

From the measurement of running nodes in a VM, it is possible to run several nodes on one machine.

Appendix A

Instructions to Compile and Run System

The source code for the simulator is hosted on GitHub. `https://github.com/KHTjessem/SwarmSI-Sim`

Before running it, it may be wise to check the settings in *main.go*. If it is configured to save all rounds, and the number of rounds is high, it will take up significant space on the drive.

This assumes you have Git and Golang installed.

To run it:

- git clone https://github.com/KHTjessem/SwarmSI-Sim
- cd src
- go run .

Virtual machine monitor script

The script developed for monitoring is available at `https://github.com/KHTjessem/VMReMon`

Bibliography

[Swarm-team(2021)] Swarm-team. Swarm. storage and communication infrastructure for a self-sovereign digital society, 2021. URL https://www.ethswarm.org/swarm-whitepaper.pdf.

[Trón(2020)] Viktor Trón. the book of swarm, 2020. URL https://www.ethswarm.org/The-Book-of-Swarm.pdf.

[foundation(2022)] Swarm foundation. The mechanics of swarm network's storage incentives, Nov 2022. URL https://blog.ethswarm.org/foundation/2022/the-mechanics-of-swarm-networks-storage-incentives/.

[Maizels(2023a)] Noah Maizels. Hive, 2023a. URL https://github.com/ethersphere/bee-docs/blob/master/docs/bee/installation/hive.md.

[Xu(2023)] Sixiao Xu. Dissecting ipfs and swarm to demystify distributed decentralized storage networks. 2023.

[Heidaripour Lakhani et al.(2022)Heidaripour Lakhani, Jehl, Hendriksen, and Estrada-Galiñanes] Vahid Heidaripour Lakhani, Leander Jehl, Rinke Hendriksen, and Vero Estrada-Galiñanes. Fair incentivization of bandwidth sharing in decentralized storage networks. *arXiv e-prints*, pages arXiv–2208, 2022.

[Maizels(2023b)] Noah Maizels. introduction.md, 2023b. URL https://github.com/ethersphere/bee-docs/blob/master/docs/learn/introduction.md.

[Gnosis(2023)] Gnosis. The community-run chain, 2023. URL https://www.gnosis.io/.

[Wikipedia contributors(2023)] Wikipedia contributors. Kademlia — Wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.

org/w/index.php?title=Kademlia&oldid=1150906957. [Online; accessed 4-June-2023].

[Raja(2023)] Haseeb Raja. Reamde.md, 2023. URL https://github.com/ethersphere/storage-incentives.

[Wikipedia contributors(2022)] Wikipedia contributors. Coordination game — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/w/index.php?title=Coordination_game&oldid=1103132746. [Online; accessed 30-May-2023].

[Gabaix(2009)] Xavier Gabaix. Power laws in economics and finance. *Annual Review of Economics*, 1(1):255–294, 2009. doi: 10.1146/annurev.economics.050708.142940. URL https://doi.org/10.1146/annurev.economics.050708.142940.

[world bank(2023)] The world bank. Metadata glossary, 2023. URL https://databank.worldbank.org/metadataglossary/gender-statistics/series/SI.POV.GINI.

[Clauset et al.(2009)Clauset, Shalizi, and Newman] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. doi: 10.1137/070710111. URL https://doi.org/10.1137/070710111.

[Medvedev(2019)] Evgeny Medvedev. Calculating gini coefficient in bigquery with sql, 2019. URL https://medium.com/google-cloud/calculating-gini-coefficient-in-bigquery-3bc162c82168.