École Polytechnique Fédérale de Lausanne

Design and Analysis of Networking Building Blocks in Malachite, a Modern BFT Consensus Engine

by Bastien Faivre

Master Thesis

Approved by the Examining Committee:

Prof. Rachid Guerraoui
Academic Advisor

Dr. Maxime Monod
External Expert

Dr. Dragos-Adrian Seredinschi
Company Advisor

February 27, 2025

If you can't commit to maintenance and support, you shouldn't write a single line of code.
— Creston Bunch

Dedicated to all the incredible people I met during my studies.

Acknowledgments

First, I want to thank my academic advisor, *Prof. Rachid Guerraoui*, for his support and guidance during this thesis and throughout my master's at EPFL, where I had the opportunity to explore the world of research in his lab.
I also thank my company supervisor, *Dr. Dragos-Adrian Seredinschi*, who guided me throughout the thesis, helping me to keep the right direction in my work.
A special thank you to *Romain Ruetschi* for helping me seamlessly onboard at the company and debug my Rust code. It was a pleasure to collaborate on my first tasks.
Moreover, I want to thank everyone I met and collaborated with at Informal. They are amazing people, and I had a great time working with them.
Besides work, I need to thank my girlfriend, *Manon*, and my friend, *Mehdi*, who supported me during times of doubt and helped me stay focused.
Finally, a huge thank you to my family, who supported me throughout my studies.

Lausanne, February 27, 2025 Bastien Faivre

Abstract

In modern blockchain-based and distributed environments, node connectivity and information dissemination are essential for robust, high-performing decentralized applications. This thesis addresses these challenges in the context of Malachite[1], a flexible, open-source, Byzantine-fault tolerant (BFT) consensus engine written in Rust. We first propose and evaluate a simple Iterative Peer Discovery (IPD) algorithm, demonstrating that it quickly establishes full connectivity in a global network of up to a few hundred nodes. Building on this, we develop a scalable, Modular Pyramidal Overlay (MPO) protocol leveraging state-of-the-art protocols to construct balanced, resilient, and low-diameter overlays. Finally, we benchmark a novel gossip protocol, Dynamic Optimal Graph (DOG) [16], against the widely used GossipSub [24], revealing significant bandwidth savings and significantly reduced CPU usage. These results confirm the viability of our methods and underline their adaptability across diverse network conditions. These building blocks help Malachite take one step further in its promise of decentralizing any applications.

[1] `https://github.com/informalsystems/malachite`

Contents

Chapter 1

Introduction

Over the past decade, blockchains have become powerful tools for decentralized finance and trustless transaction processing. Early systems such as Bitcoin[1] showcased the potential of consensus-based ledgers. Since then, the blockchain ecosystem has broadened to support use cases such as decentralized social applications (e.g., Warpcast[2]), storage (e.g., Filecoin[3]), governance (e.g., Aragon[4]), and many others. More generally, these applications are examples of distributed systems, a field whose foundations began to emerge roughly forty years ago. Among the many aspects studied, two core components ensure that nodes in a network can coordinate effectively: **node connectivity** and **information dissemination**.

Node connectivity ensures that every active participant can seamlessly join the network (i.e., connect to other participants via a mechanism called *discovery*) and maintain pathways to others, preventing any part of the system from being cut off. On the other hand, information dissemination guarantees that data, such as transactions or control signals, flow consistently to all relevant recipients. These are the high level problems we are concerned with in this thesis. At a fundamental level, every algorithm and protocol enabling these decentralized applications depends on these two building blocks to function reliably.

These components nevertheless face significant challenges in modern blockchain and distributed environments. For node connectivity, the first issue often arises during node discovery, when an arriving node must locate and establish connections with existing participants. In large or permissionless networks, naive approaches flood the system with excessive broadcast traffic or risk isolating new arrivals. Maintaining a balanced and scalable topology is likewise nontrivial, as nodes need enough redundancy for resilience without overpopulating the network with redundant links.

On the information dissemination side, the challenge involves propagating new messages (transactions, blocks, and more) across a geographically distributed set of nodes while limiting bandwidth usage. A further layer of complexity is *fault resilience*, especially under Byzantine conditions in which some nodes may behave arbitrarily or maliciously. Even in the presence of node crashes, spurious messages, or denial-of-service attempts, correct connectivity and efficient dissemination must be guaranteed.

[1] https://bitcoin.org/
[2] https://warpcast.com/
[3] https://filecoin.io/
[4] https://www.aragon.org/

Over the past decades, researchers have introduced a wide range of protocols to address node connectivity and information dissemination. For connectivity, structured overlays such as Kademlia [17], Pastry [21], or Tapestry [27] can achieve efficient lookups and bounded latencies, while unstructured designs like Gnutella rely on self-organization and fault resilience. Often, these solutions focus on optimizing a single metric, such as minimizing diameter or enhancing redundancy, which can limit their general applicability in diverse network conditions. Meanwhile, dissemination protocols continue to evolve, offering features like dynamic route blocking, probabilistic reliability, bandwidth saving, and sophisticated peer scoring. Specifically, gossip-based protocols became widely used, spreading information in a network much like rumors propagate. Many of these newer schemes have yet to be thoroughly assessed against widely deployed algorithms, revealing a gap in our understanding of how advanced methods can improve efficiency and resilience in real-world applications.

The work in the present thesis is undertaken within the context of Malachite[5], a flexible, open-source, Byzantine-fault tolerant (BFT) consensus engine written in Rust. Malachite provides a state-of-the-art Tenderming BFT consensus [3] implementation and aspires to serve as a generic library that allows developers to decentralize any application. To fully realize this vision, the networking layer must meet a high standard of robustness and adaptability in areas such as connectivity and dissemination. A single, one-size-fits-all solution is unlikely to suffice because different distributed applications exhibit widely varying workloads, network conditions, and latency requirements. Consequently, a modular and generic framework becomes essential, allowing different deployment scenarios to draw on consistent, reliable primitives while customizing details as needed.

A healthy network ultimately relies on balanced connectivity and effective data propagation. Even the most advanced consensus engines hinge on these network fundamentals to maintain liveness and consistency in the presence of crashes or adversarial nodes. Although designing generic components can sometimes introduce performance trade-offs, the development overhead they eliminate is significant. By abstracting away intricate networking concerns, integrators can direct their efforts toward application-specific logic, leveraging Malachite's underlying networking features—whether for small-scale deployments with limited resources or large-scale systems demanding top-tier throughput and resilience.

This thesis aims to address the problems of connectivity and information dissemination in distributed systems by introducing and evaluating robust, modular solutions to these building blocks. These solutions are designed to function under various deployment conditions while ensuring reliability and efficiency. The overarching objective is to integrate them seamlessly into Malachite so that this BFT consensus engine can offer a flexible yet powerful networking layer that meets diverse requirements for fault tolerance, performance, and ease of integration.

In pursuit of this goal, this thesis makes three contributions:

1. **An iterative approach to discovery**: We introduce a straightforward yet efficient algorithm tailored to networks of up to a few hundred nodes. Its focus is on minimizing overhead while maintaining full connectivity, making it an ideal foundation to evolve into more advanced techniques. We evaluate this solution in a worldwide setup to confirm its efficiency and low overhead promise.
2. **A scalable and modular P2P network overlay protocol**: We address the scalability limitations of the initial iterative approach by proposing a flexible architecture that adapts to diverse use cases, aligning with the Malachite philosophy of modularity. This protocol goes beyond discovery, maintaining a

[5] `https://github.com/informalsystems/malachite`

balanced, resilient, low-diameter network overlay. We validate a proof of concept in a 500-node network through a similar worldwide evaluation.

3. **An evaluation of a novel gossip protocol**: Dynamic Optimal Graph (DOG) [16] is a recent protocol that promises primarily bandwidth savings. We benchmark it against the well-established GossipSub [24] in a high-throughput worldwide deployment of 32 nodes. This comparison sheds light on both protocols' performance and resource consumption, providing insights into the potential of DOG as a future major gossip protocol.

This thesis is organized as follows. Chapter 2 introduces the iterative peer discovery algorithm, beginning with the necessary background and presenting the design details and an overview of the implementation. An evaluation section follows, describing the experimental setup and results and concluding with a discussion of limitations and directions for further research. Chapter 3 extends this work by proposing a scalable, modular P2P overlay protocol. As in Chapter 2, it surveys relevant background, explains the protocol architecture, covers key implementation aspects, and evaluates performance in a worldwide setting. The chapter closes with a discussion of potential improvements and avenues for exploration.

Chapter 4 shifts focus to the gossip or information dissemination layer. It provides an overview of two protocols, DOG and GossipSub, and briefly explains DOG's implementation details. The evaluation setup is then described, and the experimental results are presented. The chapter concludes by outlining how DOG could be further optimized to enhance performance in real-world deployments. Finally, Chapter 5 summarizes the thesis findings and suggests opportunities for future work across all the components studied.

Chapter 2

Discovery via an Iterative Approach

Not all networks have millions of nodes, requiring complex protocols for effective operation. This chapter presents a streamlined yet effective discovery algorithm using an iterative approach inspired by the breadth-first search (BFS) graph traversal method to discover all active nodes within a network. The algorithm is designed to achieve high efficiency and speed while maintaining minimal overhead.

First, background information about bootstrap nodes and identity validation is presented, highlighting how these considerations guided the algorithm's design. Next, the design section begins with a base algorithm and extends it to bootstrap a network from any initial configuration to a fully connected network. The final algorithm is the **Iterative Peer Discovery (IPD)** algorithm. The base and IPD algorithms and their defining properties are formally proven to hold. Following this, the chapter details the Rust implementation within the Malachite open-source codebase. The evaluation section then describes the methodology and worldwide setup used to benchmark performance when joining an existing network and spawning one from scratch. The results are presented and discussed. Finally, the chapter describes potential directions considered during the design process and outlines the algorithm's limitations regarding scalability.

2.1 Background

2.1.1 Bootstrap Nodes

The discovery process is made of 3 steps:

1. **Find bootstrap nodes**
2. **Extend knowledge**
3. **Choose peers**

The first step involves finding an initial set of bootstrap nodes with which the new node can initiate the two other steps. The second step consists of extending the network knowledge of the new node with other nodes not part of the initial set. This step results in an arbitrary view of the network determined by a specific condition, such as, for example, a full view (requiring discovery of the entire network), an *at-least N nodes*

view, or a capability-based view (ensuring knowledge of at least one peer per specified capability). Finally, from this view, the new node can decide which node keeps a connection, representing the last third step.

We refer to the initial set of bootstrap nodes as the **bootstrap set**. Also, any node can be a so-called bootstrap node; this term does not imply any additional specific behavior or feature on a node; this term is a label commonly used in the literature, and the joining node uses it to refer to the first nodes it is aware of before even starting the discovery algorithm. Moreover, note that the discovery algorithm or protocol refers to the last two steps defined before. Indeed, we will always assume that all bootstrap sets are known, i.e., all new nodes will know at least one other node in the network at the beginning, except the first node of the network, which is alone.

In a production network, Dinger and Waldhorst present [6] different mechanisms to find such bootstrap nodes:

- **Out-of-band mechanisms**: Initially achieved through Internet Relay Chat (IRC), the addresses of some active peers in a network are now shared on websites. For example, the addresses of Ethereum[1] nodes can easily be found online[2].
- **Dedicated bootstrap servers**: Many peer-to-peer (P2P) systems like BitTorrent[3] use one or more so-called bootstrap server(s) whose DNS names or IP addresses are well-known. These servers act as a network registry of active nodes, providing newly joining peers with a list of active nodes when they connect. However, it is worth mentioning that this approach keeps a kind of centralization and is subject to potential denial-of-service (DoS) attacks. Moreover, such servers could be compromised and return addresses of malicious nodes, which could lead to a risk of an eclipse attack[4] on the new nodes.
- **Local host cache**: This mechanism is not made to be used as a first-time connection to the network but rather for reconnection after a downtime (such as a crash or any disconnection types). The idea is to have nodes maintain a list of their active connections on the host machine so that, when restarting, nodes can try to reconnect to them without having to initiate a search from scratch. Note that the efficiency of this mechanism might not be guaranteed if a node has been down for a long time, as the network state might have evolved, and the previously known peers could not be valid anymore.
- **Random Address Probing**: As its name indicates, this technique sends join requests to random IP addresses and default ports. This mechanism aims to be used in massive networks, compromising millions of nodes to get a high probability of success in finding peers. The performance heavily depends on IP address range, request rate, and port standardization.
- **Network layer mechanisms and standard protocols**: Multicast, anycast, or service location protocol (SLP) can aid bootstrapping but face scalability, robustness, and global support limitations.

It is worth noticing that not all nodes in a network can be reached or used as bootstrap nodes. Indeed, in some networks, there might be two types of nodes: client and server. In the implementation of Kademlia [17] in the rust-libp2p[5] library, for example, some nodes can be configured as a client, meaning that external entities can reach out to them to execute any request on the network, but the internal protocol (such as content request, for instance) will not be able to use (or route through) these client nodes. These nodes act like *read-only* nodes and do not participate actively in the network's main functioning.

[1] `https://ethereum.org/`
[2] `https://www.ethernodes.org/`
[3] `https://www.bittorrent.com/`
[4] `https://www.ledger.com/academy/glossary/eclipse-attack`
[5] `https://github.com/libp2p/rust-libp2p`

In this thesis, we will not consider such client nodes. Ignoring them does not impact the algorithms we will present, as they do not participate in the internal protocols. We assume all nodes are exposed to the network and are reachable by everyone. Moreover, we also do not handle nodes located behind network address translation (NAT) boxes or firewalls, as this is not part of the main focus of this work.

2.1.2 Identity Validation

A perfect world does not exist; the same is true for distributed systems. We consider a **Byzantine environment**, in which Byzantine nodes are present. The latest refers to nodes that behave arbitrarily or maliciously, potentially sending conflicting or incorrect information to different system parts. The protocols developed in such an environment are labeled as **Byzantine Fault Tolerant (BFT)**.

In discovery protocols, the main challenge is identifying whether a discovered node is honest or malicious, as interacting with them could lead to catastrophes. And this is not an easy task. Indeed, Douceur states [7] that *"it is practically impossible, in a distributed computing environment, for initially unknown remote computing elements to present convincingly distinct identities. With no logically central, trusted authority to vouch for a one-to-one correspondence between entity and identity, it is always possible for an unfamiliar entity to present more than one identity, except under conditions that are not practically realizable for large-scale distributed systems."*. In such unsafe scenarios, networks are susceptible to Sybil attacks, where many malicious entities counterfeit multiple fake identities to compromise a disproportionate network share. As mitigation in the absence of such an authoritative oracle, entities use resource-demanding challenges to validate identities. Note that this kind of network is also called to be **permissionless**.

These challenges are usually referred to as **cryptographic puzzles**. Their goal is to slow down identity creation. They can also be used as a general mitigation against DoS attacks. For example, Juels and Brainard use such puzzles [12] to protect a server against depletion attacks, where a malicious actor initiates many connection requests, exhausting the server's resources. Going back to identity validation, puzzles are also controversial [5] as there is a trade-off between effectiveness and protection. Indeed, solving a puzzle should be affordable for the slowest legitimate actor. Still, it should be hard enough to slow down an attacker with sufficient resources.

Another approach typically used is a **reputation system**, a mechanism used in decentralized networks to evaluate and assign trust scores to peers based on their past behavior. These systems rely on specific metrics to assess whether a given action (such as a transaction or a message) is valid. Typically, reputation systems interact with the application layer to monitor these actions and update a peer's score accordingly. For example, in EigenTrust [13], a well-known reputation algorithm for peer-to-peer networks, each peer assigns local trust scores to others based on their history of successful or failed interactions. These scores are then aggregated across the network using a global trust computation, ensuring that peers with a high history of providing authentic data or services are considered more trustworthy. This helps mitigate malicious activity by prioritizing interactions with high-reputation peers and penalizing unreliable ones. However, a malicious node could still attack the network by first increasing its score by behaving correctly long enough to get a good score and then initiating an attack when other nodes trust it. GossipSub [24] names them *Covert Flash Attacks*. Moreover, it shows that these attacks can be costly for the attackers, adding another mitigation factor.

To keep the focus on the discovery algorithms, we decided not to include any identity validation mechanism in the presented algorithms. We argue that their absence does not impact the design of the algorithms, and they can easily be integrated afterward when deploying the protocols in a production environment. We will

still mention such integrations if they are relevant.

2.2 Iterative Peer Discovery (IPD) Algorithm Design

2.2.1 Base Algorithm

The goal of this first algorithm is to allow a new node to discover all the nodes in a network starting from an initial bootstrap set. More formally, we consider the following properties:

- **Discoverability**: The joining node eventually discovers all honest nodes.
- **Fault-tolerant**: The algorithm is resilient to Byzantine behaviors and faults, such as crashes or omissions.
- **Termination**: The algorithm eventually terminates with at least one honest peer, except for the first node of the network.

In addition to these properties, we make the following assumptions on the nodes:

- **Honesty**: An honest node always behaves as expected.
- **Byzantine**: A node diverging at least once from the expected behavior is called Byzantine. For simplicity, we include crashed and unreachable nodes due to network conditions.

And the following assumptions on the network:

- **Finite size**: The network has a finite number of nodes.
- **Connectivity**: An honest node can reach all other honest nodes via a path of honest nodes only at all times. This implies:
  - The network has no isolated honest nodes or partitions (among honest nodes).
  - An honest node has at least one honest peer if there are at least two honest nodes in the network.
- **Safe bootstrap**: There is at least one honest node in the bootstrap set of all nodes.
- **Byzantine limit**: There are at most F Byzantine nodes in the network.

The algorithm also assumes the existence of a generic request-response module with the following interface:

```
# Initialization
module RequestResponse<requestDataType, responseDataType>

# Requests
event <sendRequest, node, requestDataType>
event <sendResponse, node, responseDataType>

# Indications
event <receivedRequest, node, requestDataType>
event <receivedResponse, node, responseDataType>
event <noPendingRequest>
```

Listing 2.1: Request-response module interface.

The event `noPendingRequest` is emitted when there are no more pending requests and all received responses

have been processed. Moreover, the event is only emitted after at least one request has occurred to avoid it being emitted directly after the module's initialization.

Moreover, we suppose the request-response module to be fault-tolerant, handling the following behaviors:

- **No response to a request**: the module handles such cases using a request timeout or failure detector if the contacted node is down. This is mainly an implementation design, but we assume a request will never wait infinitely. This situation is referred to as an involuntary crash or omission fault, or to voluntary Byzantine behavior.
- **Request made to invalid node**: if we attempt to dial invalid nodes, the module leverages the same mitigation technique as for the previous behavior. This case is classified as a network fault.

With all these building blocks, we can now define the base discovery algorithm interface:

```
# Initialization
module BaseDiscovery<>
uses:
    RequestResponse<_, set<node>> # no request data

# Request
event <start, set<node>> # the set is the bootstrap set

# Indication
event <done, set<node>>
```

Listing 2.2: Base Discovery module interface.

And the base algorithm itself:

```
# Node's local variables
local_peers = set<node>{}
contacted = set<node>{}

upon event <start, S>:
    for node in S:
        trigger <sendRequest, node, _>
        contacted.insert(node)

upon event <receivedRequest, node, _>:
    trigger <sendResponse, node, local_peers>
    local_peers.insert(node)

upon event <receivedResponse, node, peers>:
    local_peers.insert(node)
    for peer in peers:
        if not contacted.contains(peer):
            trigger <sendRequest, peer, _>
            contacted.insert(peer)

upon event <noPendingRequest>:
    trigger <done, local_peers>
```

Listing 2.3: Base Discovery module algorithm.

Note that the terms node and peer are used interchangeably for readability, as they refer to the same entity type.

The algorithm aims to discover nodes with an iterative approach similar to the breadth-first-search (BFS) graph algorithm. A node will contact all peers in its bootstrap set and ask them for their known peers. The node then repeats the process with the newly discovered peers until no more peers are discovered.

2.2.2 Proof

First, note that a node processes events sequentially. We prove each property independently:

- **Discoverability**: By the *safe bootstrap* assumption, the joining node receives at least one response (from an honest node) with a set of peers, among which, by the *connectivity* assumption, there is at least one honest node (except if this is the only honest node actually in the network). By iteratively contacting every received peer and applying the *connectivity* assumption, and by the fact that a node is never contacted twice with the contact set, the algorithm eventually discovers all honest nodes.
- **Fault-tolerant**: Here are the possible behaviors and their mitigation:
  - **No response to a peer request**: mitigated by the fault-tolerant design of the request-response module.
  - **Request made to invalid node**: mitigated by the fault-tolerant design of the request-response module.
  - **Response with a massive amount of nodes (likely invalid)**: the response is finite by the *finite size* network assumption; therefore, the previous behavior's mitigation handles this scenario too.
  - **Flood the local peers set with many requests from different identities**: this scenario is possible since, when receiving a request, a node will add the requesting node to its local peers set. The node should verify the identity of the node to ensure it is valid. We ignore this attack vector based on the identity validation discussion 2.1.2.
- **Termination**: By the two previously proved properties, the *finite size* network assumption, and the `noPendingRequest` event of the request-response module, the algorithm eventually terminates.

Regarding complexity, the total number of messages is linear in the total number of nodes N, namely $2N$ (request and response to each node). However, considering the biggest Byzantine response being of size L (finite by the *finite size* network assumption), there can be at most $F \cdot L$ additional requests to invalid nodes, leading to a total message complexity of $O(N + F \cdot L)$.

2.2.3 Limitation

Despite allowing a new node to discover the entire network, the current state of the algorithm does not allow bootstrapping a fully connected network by spawning all nodes concurrently. Indeed, suppose the following 5-node network with the following bootstrap sets:

	node A	**node B**	**node C**	**node D**	**node E**
Bootstrap set	B	C	D	E	A

Table 2.1: Bootstrap sets of a 5-node network.

Consider the execution where all five nodes contact their respective bootstrap node concurrently. When

receiving the request, each node adds the requesting peer to their local peers set and sends back an empty set as they did not have anyone in their local peers set yet. Upon receiving the empty set, the request-response module of each node detects that there are no more pending requests, and since all received responses were processed, the discovery algorithm terminates with an incomplete network:

	node A	**node B**	**node C**	**node D**	**node E**
Discovered peers	B, E	A, C	B, D	C, E	A, D

Table 2.2: Discovered peers of each node.

This resulting network might be sufficient in some cases, but one might want a fully connected one. There are multiple trivial solutions to this problem, such as deferring the spawn of each node, using the same bootstrap node for everyone, or giving a complete nodes list as a bootstrap set to everyone. But all of them rely on quite strong assumptions. The following algorithm tackles this problem.

2.2.4 Bootstrap Extension

This algorithm extension aims to achieve a fully connected network on concurrent execution with minimal configuration and constraints. First, let's consider the directed graph obtained with the bootstrap sets where an edge (u, v) means that u has v in its bootstrap set. For example, here is the graph of the example given in Table 2.1:

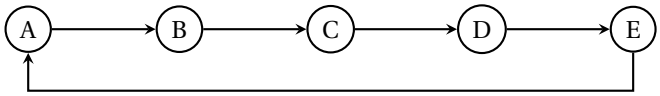


Figure 2.1: Bootstrap graph of 5 nodes.

This specific graph has a valuable property: it is **strongly connected**. It means that, for each pair of nodes u, v, a path exists from u to v in the directed graph.

In contrast, a graph where each node can reach the other nodes if we ignore the edges' direction is called **weakly connected graph**:

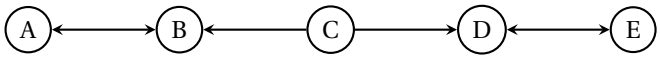


Figure 2.2: Weakly connected graph.

Finally, here is a **disconnected graph**:

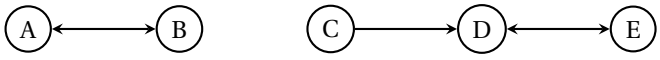


Figure 2.3: Disconnected graph.

We will not consider this last case simply because it is impossible for two distinct partitions to learn about each other, except if we give at least one node from one partition the information about at least one node

in the other partition. But that would mean that there is an edge between these two nodes, and hence, we would not have any partitions, leading to a weakly connected graph. Moreover, since we want to design an algorithm with the least information possible, we will consider that the only information known about the bootstrap sets is that its associated graph is weakly connected.

In addition to the defined properties of the base algorithm at 2.2.1, we add the following:

- **Concurrency**: The algorithm can be executed on many nodes concurrently, and they all eventually discovery each other, leading to a fully connected network.

Note that we consider all nodes to start being responsive (alive) simultaneously, but they will differ in their execution speed. Otherwise, some nodes might contact other nodes that do not respond despite being honest, and it does not make sense in the context of this algorithm.

Here is the **Iterative Peer Discovery (IPD)** algorithm interface:

```
# Initialization
module IPD<>
uses:
    RequestResponse<set<node>, set<node>>

# Request
event <start, set<node>> # the set is the bootstrap set

# Indication
event <done, set<node>>
```

Listing 2.4: IPD module interface.

And the algorithm itself:

```
# Node's local variables
local_peers = set<node>{}
contacted = set<node>{}
bootstrap_nodes = set<node>{}

def contact(peers):
    for peer in peers:
        if not contacted.contains(peer):
            # 'U' is the set union operator
            trigger <sendRequest, peer, local_peers U bootstrap_nodes>
            contacted.insert(peer)

upon event <start, S>:
    bootstrap_nodes = S
    contact(bootstrap_nodes)

upon event <receivedRequest, node, peers>:
    # '\' is the set difference operator
    trigger <sendResponse, node, (local_peers U bootstrap_nodes) \ peers>
    local_peers.insert(node)
    contact(peers)

upon event <receivedResponse, node, peers>:
    local_peers.insert(node)
    contact(peers)
```

```
26
27 upon event <noPendingRequest>:
28     trigger <done, local_peers>
```

Listing 2.5: IPD module algorithm.

The addition to the algorithm allowing the bootstrap capability is the **full-knowledge sharing** on both request and response. Indeed, the idea is to force each set of two peers to share their neighbors so that they end up with the same network view at that time.

2.2.5 Proof

First, the **Fault-tolerant** property proof is the same as the base algorithm proof at 2.2.2. We now prove the **Discoverability**, **Termination**, and **Concurrency** properties together.

Let $G = (V, E)$ be the directed graph of the bootstrap sets, where V is the set of nodes and $E \subset V \times V$ is the set of directed edges between the nodes. We assume that the graph G is at least weakly connected. We can generalize the connections of a node $u \in V$ as shown in Figure 2.4. We define the *predecessors* of a node u the nodes a_i, $1 \leq i \leq n$ such that $(a_i, u) \in E$. Moreover, we define the *successors* of a node u the nodes b_j, $1 \leq j \leq m$ such that $(u, b_j) \in E$. Remember that a node may have no predecessors or successors but not both since isolated nodes are not considered.

We now define two lemmas.

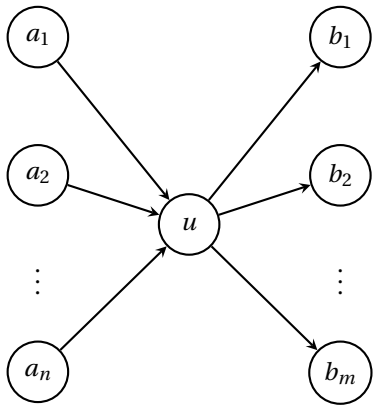


Figure 2.4: Connections of a node.

Lemma 1 (Predecessors discoverability)**.** *Every node a_i eventually discover all nodes a_j, $j \neq i$ and node u.*

Proof. When node u receives a request from a node a_i, it responds to the node with its current set of local peers and then adds the node a_i to it. By the sequential processing of requests on node u, a given node a_i will receive an answer with all a_j, $j \neq i$ whose request was processed before the one of a_i. Moreover, a_i will be included in the response of u to all a_k, $k \neq i$, $k \neq j$ whose request was not processed by u yet. Therefore, by the algorithm, node a_i eventually contacts all nodes a_j and will be eventually contacted by all nodes a_k. And

since a successful request-response between two nodes implies that each adds the other to its local peers set, every node a_i eventually discovers all nodes $a_j,\ j \neq i$ and node u. □

Lemma 2 (neighbors discoverability)**.** *Every node n_i that is a predecessor or successor of node u eventually discovers all nodes $n_j,\ j \neq i$ and node u.*

Proof. By the *predecessors discoverability* lemma, all nodes a_i eventually discover each other. Moreover, by the bootstrap set sharing in the response from the algorithm, all nodes a_i eventually discover all nodes b_j. Finally, by the bootstrap set sharing in the request from the algorithm, all nodes b_j eventually discover each other. Therefore, every node n_i that is a predecessor or successor of node u eventually discovers all nodes $n_j,\ j \neq i$ and node u. □

Corollary 3 (neighbors identical network view)**.** *Lemma 2 implies that all nodes n_i eventually have the same network view due to the bootstrap set sharing during the communications between all nodes.*

We now show that this principle extends to the complete graph and leads to everyone eventually discovering each other.

Take any node $u \in V$; by the *neighbors discoverability* lemma, we eventually obtained a fully connected graph C_1 of connections with its neighbors. Then, take any node $v \in C_1$ that has some neighbors $n_v \notin C_1$ (i.e. $n_v \subseteq V \setminus V_{C_1}$). By the *neighbors discoverability* lemma and its corollary, all nodes from C_1 eventually discover all n_v. We call the new resulting graph C_2 (note that $C_1 \subset C_2$). Then, we continue the process, each time taking a node $w \in C_i$ that has neighbors $n_i \notin C_i$, and we apply the *neighbors discoverability* lemma and its corollary to get C_{i+1}.

Then, by iteratively continuing the process, and since $C_i \subset C_{i+1}$, we eventually get the complete, fully connected graph.

2.3 Implementation

The IPD algorithm is implemented[6] using the rust-libp2p[7] library, a modular peer-to-peer networking framework. To simplify things, consider the library's two components relevant to the context: the **swarm** and the **behavior**. The swarm contains the state of the network and defines how it should behave based on a given behavior. This is the entity from which the events are polled. We define so-called *swarm events*, such as peer connection, disconnection, dial error, etc. Then, the behavior is built with the chosen protocol we want to use, and each of these protocols compromises its behavior with its events that are also polled via the swarm. Note that all events are sequentially processed one by one by polling the swarm; there is no concurrent event processing.

For the IPD algorithm, we define a behavior with the following two protocols:

[6] `https://github.com/informalsystems/malachite/tree/87511a77860cc9b86c9597b5c71c0fa08eada031/code/crates/discovery`

[7] `https://github.com/libp2p/rust-libp2p`

- **identify**[8]: This protocol exchanges peer information (such as addresses, public keys, capabilities, etc.) between peers. In the algorithm, we do not consider a connection to be a peer before exchanging this information.
- **request-response**[9]: This protocol implements a simple request-response mechanism, matching the request-response module we defined in the design section.

Regarding the interaction between nodes, it is worth mentioning that communication is made of two steps. The first one is called *dialing* and is used to open a connection. Then, the protocols (second step) can do their job on top of this connection. Therefore, we define a so-called **Handler** to handle all these steps that will keep track of dials and requests. More specifically, this handler module can be fine-tuned with the following configuration variables:

- `REQUEST_TIMEOUT` defines the timeout for a request from the request-response protocol. The default is 5 seconds.
- `MAX_CONCURRENT_DIALS` defines the maximum number of dial attempts that can be made concurrently. The default is 2.
- `MAX_CONCURRENT_REQUESTS` defines the maximum number of request attempts that can be made concurrently. The default is 2.
- `MAX_DIAL_RETRIES` defines the maximum number of times a dial can be retried. The default is 5.
- `MAX_REQUEST_RETRIES` defines the maximum number of times a request can be retried. The default is 5.

Moreover, the retry mechanism implements a Fibonacci backoff for the delay between two attempts, with an initial delay of 1 second.

2.4 Evaluation

2.4.1 Methodology

We evaluate the join and spawn performance of the IPD algorithm in both local and distributed setups.

- **Join**: We measure the time required for a node to join a fully connected network of up to 500 nodes. The IPD algorithm's performance (with a single-node bootstrap set) is compared to an *optimal* discovery scenario, where the joining node has a bootstrap set containing the addresses of all existing nodes. In the distributed benchmark, we also examine how varying the number of concurrent dials and requests affects performance.
- **Spawn**: We measure the time needed for a network of up to 256 nodes to become fully connected. We also investigate the effect of bootstrap set size on performance. The results are also compared to an *optimal* spawn scenario, where each node has a bootstrap set containing the address of the previous $N/2$ nodes in a cyclic list of indices[10].

For both *optimal* reference cases, the IPD algorithm is disabled, and discovery consists solely of connecting

[8] `https://docs.rs/libp2p/latest/libp2p/identify/index.html`
[9] `https://docs.rs/libp2p/latest/libp2p/request_response/index.html`
[10] For $N = 11$ for instance, node three will contact nodes $(0,1,2,9,10)$ and will be contacted by nodes $(4,5,6,7,8)$.

to predefined bootstrap nodes.

For the local setup, we use a Hetzner[11] cloud machine (CCX43) with 16 dedicated vCPUs (AMD EPYC) and 64GB of RAM, located in the Nuremberg (eu-central) data center.

For the distributed setup, we use Hetzner cloud machines (CCX33) with 8 dedicated vCPUs (AMD EPYC) and 32GB of RAM, deployed across Hillsboro (us-west), Nuremberg (eu-central), and Singapore (ap-southeast). Table 2.3 provides the round trip times (RTTs) between these regions.

	us-west	**eu-central**	**ap-southeast**
us-west	0	173	187
eu-central	173	0	178
ap-southeast	187	178	0

Table 2.3: RTT (ms) between data centers.

Regarding the execution of the join benchmark, we start one node at a time, and we wait for network stabilization (i.e., the previous node is done with its discovery) between each spawn. Moreover, for the distributed setup, the nodes are spawned on all locations in a round-robin fashion.

For the spawn benchmark, all nodes start simultaneously, and we wait until all nodes are done with discovery. Moreover, we had to set the concurrent factors to a low value 2. Indeed, since all nodes send traffic simultaneously, the workload was too heavy with higher values, leading to some nodes crashing.

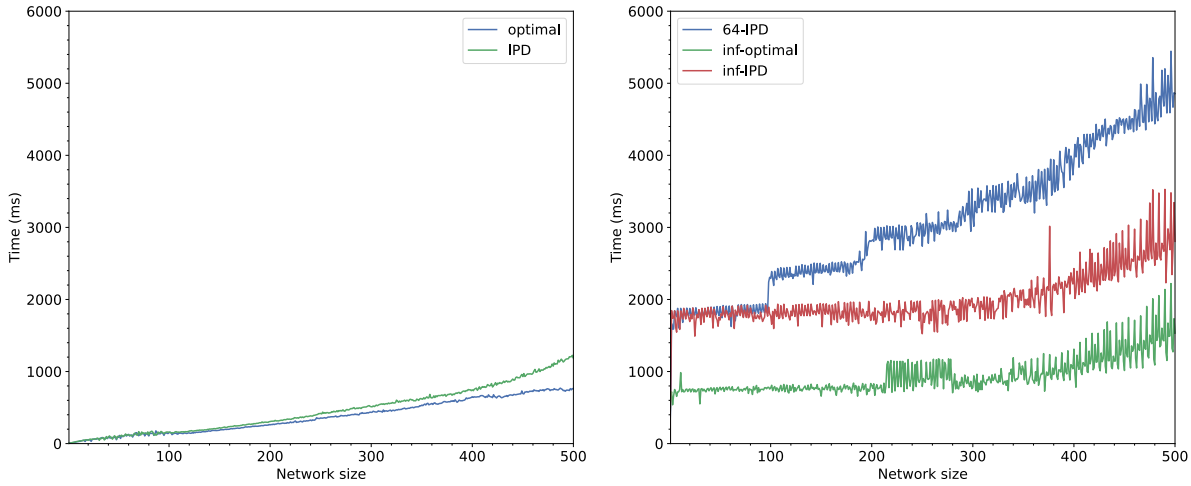
2.4.2 Results

Figure 2.5a shows that the performance of the IPD algorithm is quite close to the optimal discovery. The gap represents the step of reaching out to the bootstrap node to retrieve all other peers before connecting to them. Moreover, the results highlight a performance degradation starting from a network of size about 420. This drop is due to the machine's resources reaching their limits.

Figure 2.5b highlights that the IPD algorithm takes slightly more than twice the time of the optimal discovery. The main reason is the latencies between machines. Indeed, the additional steps executed bring a latency overhead, increasing the final discovery time. Moreover, we observe jumps in the performance when we restrict the algorithm with 64 concurrent dials and requests. Remember that the setup consists of 3 machines; therefore, when joining the network, a new node will have two-thirds of the nodes to contact located on the two other machines. Due to the 64 concurrency constraint, the algorithm will need additional steps when contacting more than 64 nodes remotely, leading to jumps every 96 nodes.

We observe in Figure 2.6a a fast setup for small networks. However, as the size increases, the performance degrades due to the quadratic complexity of the generated communication traffic. Moreover, all this traffic is handled by a single machine, which quickly reaches its upper bound limit. The impact of high traffic is highlighted in Figure 2.6b with the biggest bootstrap set size, where the probability of two nodes contacting each other (and therefore generating twice the necessary traffic) is higher. Overall, the fastest spawn is still 31% slower than the optimal spawn.

[11] `https://www.hetzner.com/`



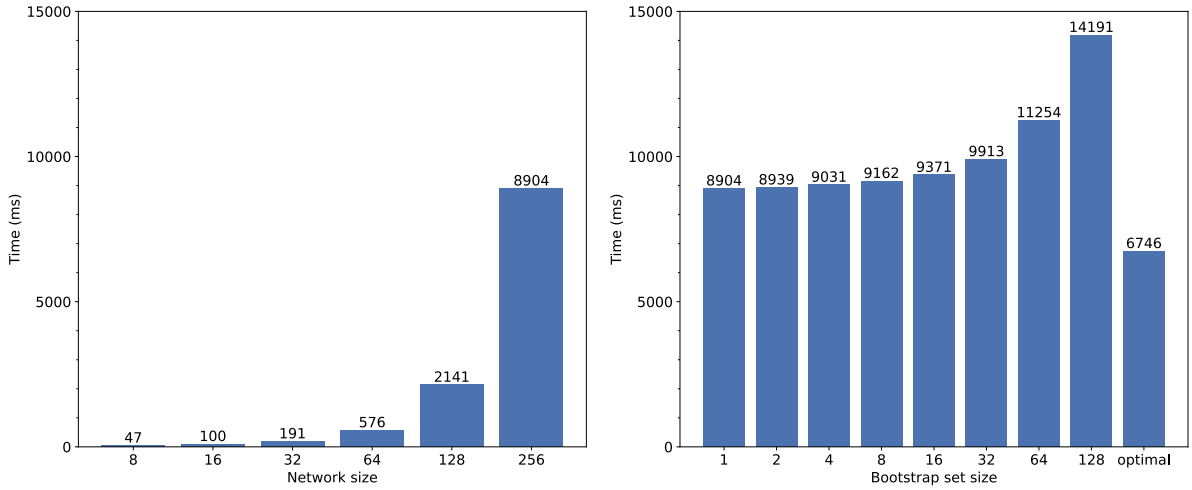
(a) Local.

(b) Worldwide.

Figure 2.5: Time to join a network.

Figure 2.7a clearly exposes the impact of latency and concurrent factors on performance, even for small networks. It is also worth noticing that the machines' CPUs were not overwhelmed. This resource availability makes bigger bootstrap sets perform better, but there is still a balance to keep, making 16 the ideal size. However, the IPD algorithm is still about 42% slower than the optimal spawn.

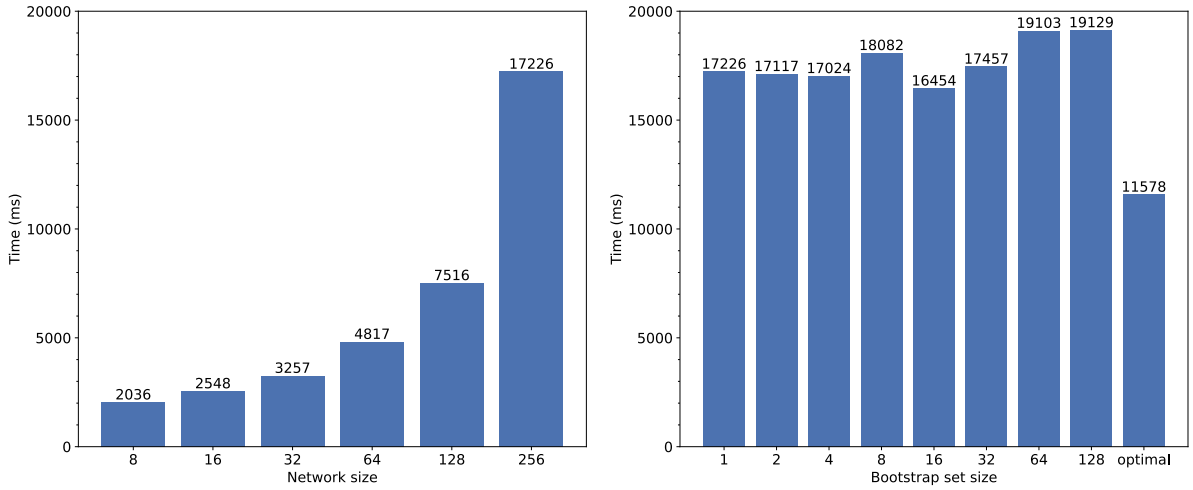
31% and 42% seem to be consequent losses in performance, but it is worth mentioning that the IPD algorithm has, in the worst case, twice as many communication steps to make. Overall, the spawn time remains reasonable.



(a) Time to spawn a local network.

(b) Time to spawn a local 256-node network.

Figure 2.6: Time to spawn a local network.



(a) Time to spawn a worldwide network.

(b) Time to spawn a worldwide 256-node network.

Figure 2.7: Time to spawn a worldwide network.

2.4.3 Remarks

While executing the benchmarks, we observed variations in the results depending on whether TCP or QUIC was used as the transport layer. However, since this was not this thesis's primary focus, we omitted this part of the study. However, note that we only compared runs using the same protocol.

Furthermore, although the spawning feature is interesting, it is unlikely that the IPD algorithm would be used to spawn a network. If the goal is to spawn a network of a specific size, the optimal solution would likely be preferred for faster execution and lower resource consumption. We conducted this benchmark to assess the algorithm's performance and overhead.

Regarding the join benchmark, since nodes were added progressively to the network, you may have noticed that the IPD algorithm consists of only two steps when joining a fully connected network. First, the bootstrap node is contacted to retrieve the addresses of all other nodes, and then a connection is established with each of them. More generally, the performance of the IPD algorithm in a random network can be assessed by noting that the number of steps corresponds to the longest diameter of the network from the bootstrap node.

2.5 Discussion

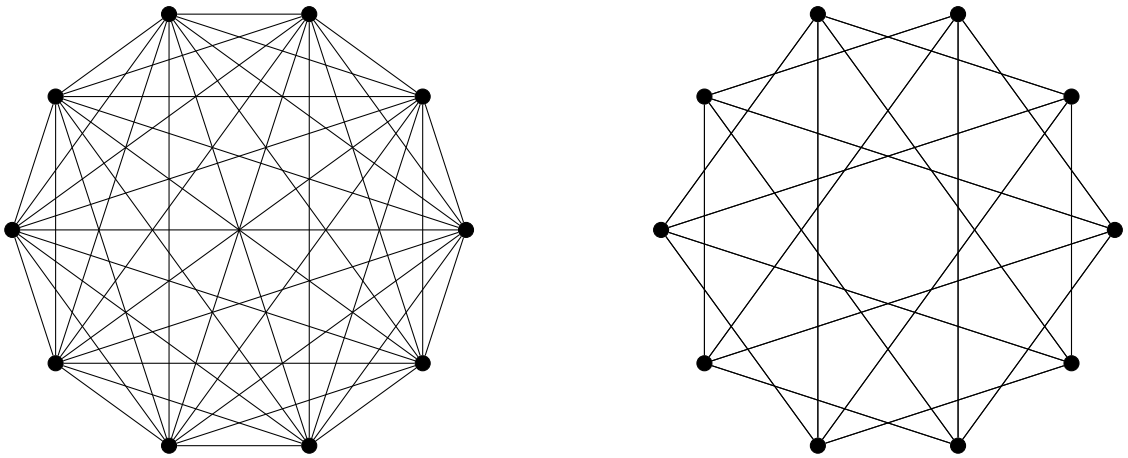
When designing the algorithm, we had the choice between an iterative and a recursive approach. The recursive approach consists of the new node sending a join message to its bootstrap nodes, and the latter will forward the join message to their neighbors recursively. When receiving a join message, a node would contact the new node to announce itself. A similar idea is used in Pastry [21], where the nodes respond to the join message with their state tables.

We decided to stick to the iterative approach because it better tracks the progress of discovery. Indeed, by contacting the nodes by itself, the joining node can more easily detect the end of the discovery protocol. On

the other hand, with the recursive approach, the joining node is passive to the protocol. It could potentially be contacted by another node at any time, making the progress tracking a bit messy.

2.6 Limitations

We focus here on the discovery capability of the IPD algorithm (and not the spawn). An algorithm aiming to discover everyone does not scale. Indeed, imagine a network with millions of nodes; a single node could not handle that many peers. Moreover, a node might want more control over its connections for many reasons, such as limited resources or a minimum resilience level. More specifically, what if a node could control its inbound (initiated by someone else) and outbound (self-initiated) connections?



(a) Fully connected.

(b) out-2-in-2 restricted.

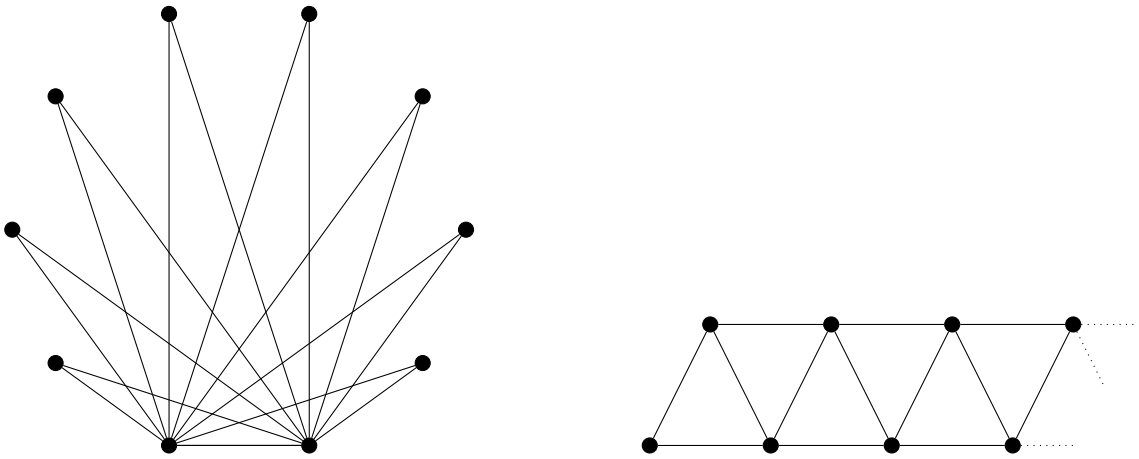
Figure 2.8: 10-node network.

Consider a fully connected network as shown in Figure 2.8a, and suppose that each node wants to restrict its connections to two inbound and two outbound connections; one would ideally think to obtain a network such as shown in Figure 2.8b. But, as we just mentioned, this is an ideal scenario. Indeed, in a real-life scenario, not all nodes would necessarily have the same constraints in terms of connections, and we could obtain so-called *risky* networks, as shown in Figure 2.9.

The network in Figure 2.9a has a centralized design that makes it highly sensible to faults. Indeed, the entire network connectivity relies on the two bottom nodes. If these two nodes were to crash or stop, then the network would become partitioned.

In addition to a similar fault sensibility, the network in Figure 2.9b also presents a performance issue due to its high diameter. Indeed, information would need to travel through many nodes before reaching everyone, causing potentially important issues for the application protocol operating on top of this network.

It is worth noticing that there are trade-offs between the network diameter and the total number of connections. Indeed, a fully connected network has a diameter of 1, but it increases as we reduce the number of connections. Moreover, as we want to avoid a centralized structure, it also implies a bigger resulting network



(a) Centralized network.

(b) High diameter network.

Figure 2.9: Risky networks.

diameter.

Moreover, note that we are now not only talking about discovery anymore. Indeed, with the willingness to scale and reduce the connections comes a new challenge: **overlay maintenance**. Indeed, the resulting network topology is referred to as an overlay that continuously needs to be kept healthy. It should dynamically adapt to churn while maintaining a balanced topology. The next chapter presents such a network overlay.

Chapter 3

P2P Network Overlay Protocol via a Scalable and Modular Approach

At the end of the previous chapter, we discussed the scalability issues of the full discovery and introduced the concept of overlay maintenance. This chapter describes a scalable and modular protocol compromising node discovery and churn handling to obtain a balanced overlay. Moreover, the design of this protocol is engineered with the Malachite philosophy in mind, meaning that it aims to be adaptable to any use case.

Following a similar chapter structure as the previous one, we start with background information about structured P2P networks, with a detailed focus on Kademlia [17], as we will be using this protocol as proof of concept. Then, the design section presents the pyramidal structure of the protocol called **Modular Pyramidal Overlay (MPO)**. Next, implementation details are mentioned as they are a key point of the modularity of the protocol. Then, we present an evaluation of the proof of concept leveraging the Kademlia protocol as a basis. Finally, the chapter concludes by discussing potential future research related to this topic.

3.1 Background

3.1.1 Structured P2P Networks

Peer-to-peer (P2P) networks have evolved significantly over the past two decades. Early unstructured P2P systems, such as Gnutella, exemplify a simple yet highly robust way for nodes to discover and connect with one another. These networks characteristically exhibit a low-diameter property as they often follow a power-law degree distribution [20], indicating that a relatively small number of nodes act as highly connected hubs while most nodes have far fewer connections. The formation of these hubs naturally emerges from the way new nodes join the network: they usually connect to already well-connected nodes, increasing their central importance in the overall topology.

Over time, attention shifted to structured P2P networks, introducing an identifier (ID) space for nodes and constructing an overlay based on these IDs. Popular examples, known as **Distributed Hash Tables (DHTs)**,

include well-known protocols that map data objects to node identifiers to ensure efficient lookups. Here are some of the well-known DHT implementations with a focus on how they maintain the network overlay:

- **Pastry** [21] uses a prefix-based routing table, whose information ensures efficient maintenance of the network overlay. When a new node joins, it initializes its leaf set, routing table, and neighborhood set by routing a *join* message to the closest existing node with the most similar ID. Affected nodes update their state accordingly. When a node fails or departs, its absence is detected through failed communication, and its entry in the leaf set or routing table is replaced by querying nearby live nodes. The neighborhood set is periodically refreshed to maintain locality properties, ensuring Pastry's resilience.
- **Chord** [22] employs a distributed hash table (DHT) with a ring-based overlay, where each node maintains a successor list and a finger table for efficient lookups. When a new node joins, it determines its position in the ring by querying existing nodes and updating its successor and predecessor pointers. It then transfers relevant key-value pairs and adjusts finger tables accordingly. When a node leaves or fails, its successor takes over its responsibilities, and affected nodes update their finger tables. Periodic stabilization ensures consistency, maintaining efficient lookups and fault tolerance.
- **Tapestry** [27] utilizes a prefix-based routing table and a decentralized object location system to maintain the overlay structure. When a new node joins, it initializes its routing table by querying nearby nodes and updates its neighbor and *backpointer* tables. The node integrates into the overlay by adjusting its surrogate routing entries and informing affected nodes. When a node fails or departs, its absence is detected through failed communication, triggering neighbor and *backpointer* repairs via alternate routes. Periodic maintenance ensures routing integrity, enabling efficient message delivery and fault tolerance.
- **Content Addressable Network (CAN)** [19] structures its overlay using a multi-dimensional coordinate space for decentralized key-value storage and routing. When a new node joins, it discovers an existing node, splits its assigned zone, and updates the routing table of affected neighbors. It then replicates relevant key-value pairs and informs surrounding nodes of the overlay change. When a node leaves or fails, its zone is merged with a neighbor or reassigned, ensuring data availability. Routing tables are updated dynamically, and periodic maintenance enhances resilience, allowing CAN to sustain efficient lookups.
- **Kademlia** [17] uses an XOR-based distance metric to structure its DHT, maintaining k-buckets for efficient routing, node discovery, and fault tolerance. This implementation is detailed in the next section.

Although these approaches successfully guarantee scalability and efficient routing under ideal conditions, some argue that relying on a predetermined ID space only loses the self-organizing nature that unstructured P2P networks have. This critique led to innovative hybrids such as P-Grid [1], which blend the spontaneous nature of unstructured systems with the systematic organization of structured overlays.

Meanwhile, other protocols explored topologies in which specific nodes are deliberately given more importance, reflecting real-world conditions where some participants have greater resource capabilities. For instance, PLANES [23] proposes leveraging powerful nodes (referred to as *altruists*) as a fully connected subnetwork to which other nodes attach, forming a cluster-based overlay. Similarly, Phenix [26] justifies a similar layout by analogy to social media *stars*, which naturally attract the majority of connections. SWOP [9] follows this pattern by designating nodes into head- and inner-nodes, and LDEPTH [18] employs a two-level hierarchy using a linear diophantine equation (LDE) to organize nodes.

Beyond these hierarchical or hybrid structures, several protocols showcase more unconventional or *artistic*

designs. T-MAN [11] constructs overlays that resemble torus-like topologies; BATON [10] adopts a balanced tree structure to manage peer connections; and SCAN [8] employs weighted edges in a manner inspired by biological neural connections. Another branch of research, known as locality-aware P2P protocols, endeavors to improve performance by connecting nearby nodes in the underlying network. A paper [25] compared different network-based metrics and shows that *intra-AS* clustering can benefit both ISPs and P2P users. Moreover, they state that multi-level and multi-metric neighbor selection strategies can lead to better performance.

It is important to note that many of these systems address the challenge of data placement and retrieval (as in a classic DHT). Still, our focus here remains solely on how the overlay is formed and maintained. In the context of Malachite with genericity in mind, one might ask how to choose the *best* topology among such diverse designs. Each approach exhibits distinct advantages depending on network conditions, node capabilities, or desired properties (like fault tolerance, latency, or simplicity). The possibility arises: rather than committing to a single overlay design, why not build upon them interchangeably, depending on the current use case?

Indeed, the later described protocol in this chapter takes inspiration from a paper stating [15] *"...[SWOP] was built on the top of the existing structured P2P networks..."*. Our approach similarly leverages a common observation: regardless of the specific design, each peer maintains some sets of neighbors organized according to specific metrics (e.g., distance, latency, or identifier ranges). The approach can integrate any most appropriate overlay depending on the requirements by exploiting this inherently *sorted* view in each node's neighbor set.

3.1.2 Kademlia

Kademlia is a DHT introducing a unique routing and node discovery approach, leveraging an XOR-based distance metric to structure its overlay network. This design enables efficient, low-latency lookups while minimizing maintenance overhead. Unlike earlier DHTs, Kademlia naturally propagates routing information as a side effect of query operations, reducing the need for explicit updates. The system supports parallel, asynchronous queries to improve the fault tolerance of the network and mitigate delays caused by failed nodes. Additionally, Kademlia's routing table structure provides flexibility in query routing, allowing nodes to optimize for low-latency paths. These properties collectively enhance the network's scalability, efficiency, and resilience, making Kademlia a robust foundation for modern P2P applications. We will focus here on the parts of Kademlia related to overlay maintenance, not storage capability.

Like other DHT implementations, Kademlia assigns an ID to each node; an ID is a 160-bit identifier, as well as for data keys (where one is likely to use SHA-1 hash on the data to get the key). Moreover, nodes are treated as leaves of a binary tree. The distance metric used is the exclusive or (XOR) binary operator; we define $d(x, y) = x \oplus y$ with the following properties:

$$d(x,x) = 0 \quad , \quad d(x,y) > 0 \text{ if } x \neq y \quad , \quad d(x,y) = d(y,x) \quad , \quad d(x,z) + d(z,y) \geq d(x,y)$$

For example, the distance between the 4-bit identifiers 1010 and 1100 is ${}_b1010 \oplus_b 1100 =_b 0110 = 6$.

The routing table of Kademlia is made of so-called k-buckets. Each k-bucket contains nodes with which the local node has a given distance. Figure 3.1 highlights the k-buckets (in blue) for a node. Note that this binary tree represents the distances with the local node, not the key space! Therefore, the local node is located at the left-most node (i.e., distance zero). They are B k-buckets, where B is the number of bits of the identifiers, and

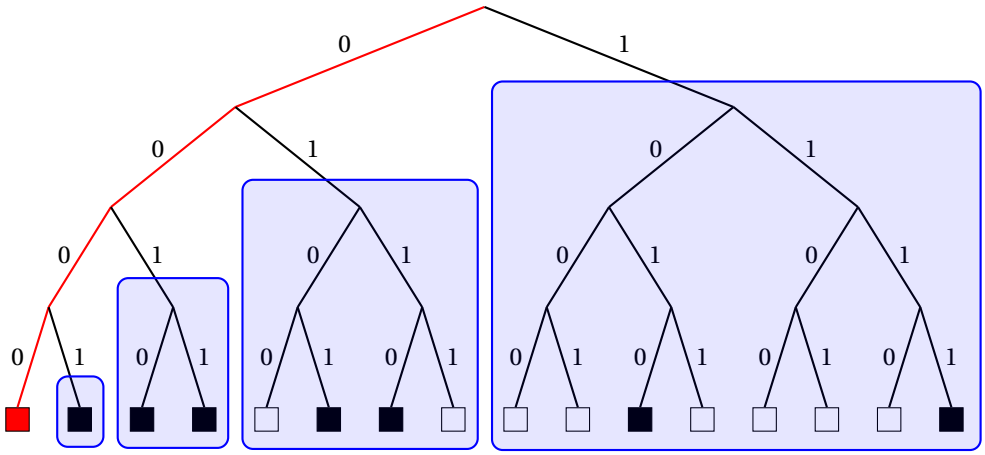


Figure 3.1: The k-buckets of a node with $k = 2$ and 4-bit identifiers.

k-bucket i ($0 \leq i < 160$) contains nodes whose distance d to the local node is $2^i \leq d < 2^{i+1}$. Moreover, with 160-bit identifiers, storing everyone would not be convenient. Therefore, Kademlia defines a parameter k that sets the maximum size of a k-bucket (typically $k \simeq 20$). For example, Figure 3.1 also mentions possible k-buckets (selected nodes in black) with $k = 2$.

With such a size constraint comes an eviction policy for each k-bucket. First, the k-buckets are sorted by the least recently seen (head); a new node is inserted at the tail of the corresponding k-bucket. If the k-bucket is full, the least recent node is pinged. If it fails to respond, it is evicted from the k-bucket, and the new node is inserted at the tail. Otherwise, the pinged node is moved back to the tail, and the new node is discarded. This preference for old contact is based on a study in the paper stating that a node that has been up for a long time is likely to stay up in the next hour. Moreover, it also adds DoS protection by mitigating routing flooding. The new nodes will only be inserted when the old ones leave the network.

The two RPCs relevant here are `PING` and `FIND_NODE`. The `PING` RPC checks whether a node is still alive, and the `FIND_NODE` RPC aims to find the k closest nodes to a given identifier. The latter is also called *node lookup*; it initiates a parallel request to α nodes from the closest non-empty k-bucket (or the α closest node) to the given identifier. Despite being called *recursive* in the paper, the protocol proceeds iteratively. Based on the previous request responses, it continues sending requests to even closer nodes until it gets a response from the k closest node it encountered during the protocol. Note that the behavior is similar to the IPD algorithm from the previous chapter.

The node lookup protocol is mainly used in the bootstrap process of a new node. Indeed, the new node starts by querying its own identifier in the network to find nodes close to it. Moreover, the joining node initiates a query for each k-bucket with a random identifier part of them. This process allows the new node to populate its routing table and to be inserted into other nodes' k-buckets if possible.

3.1.3 Discv5

During our research, we came across **Discv5**[1] which is a node discovery protocol used in Ethereum. The system is based on Kademlia, where the stored data are Ethereum node records (ENR). The DHT acts as a database of alive nodes and allows enumeration of the network by *walking* the DHT. Moreover, it presents an authoritative resolution of node records, meaning that the most recent version of a record can be retrieved. Finally, one of the most interesting features is **topic advertisement**: a node can advertise a topic, and other nodes can find it.

Despite being functional, the protocol does not automate any behavior because it is the node's responsibility to choose what to do with the database and choose with which node it wants to interact. This is primarily what we want to achieve with the overlay maintenance. However, we will discuss the topic of advertisement capability in this chapter's Discussion section 3.5.

3.2 Modular Pyramidal Overlay (MPO) Protocol Design

3.2.1 Architecture

We want to design a scalable and modular protocol for a P2P overlay. We define the following properties:

- **Scalability**: The protocol should scale to any network size.
- **Self-organization**: The protocol automatically bootstraps and maintains the network overlay. This includes node discovery and churn handling.
- **Modularity**: Both previous properties depend on the chosen organizational primitives. They should hold whatever the primitives.
- **Degree Control**: each node can control its desired number of connections.

As introduced in the previous section, we present here a design leveraging the organization paradigm of the state-of-the-art (SOTA) protocols. More specifically, we build a protocol on top of them called **Modular Pyramidal Overlay (MPO)**.

Figure 3.2 presents the pyramidal construction of the MPO protocol. We first define **virtual connections** (i.e., not opened connections), where the **Physical** layer represents the actual Internet network, visualized as a virtual, fully connected network, and the **SOTA** layer serves as the basis overlay later used for the final overlay construction, organized based on the chosen SOTA protocol. Then, based on the SOTA overlay, **direct connections** are established, where the **Selection** (final) overlay is formed by selecting specific connections from the SOTA overlay using an arbitrary selection algorithm, and the **Application** layer represents the actual instance operating with the selected connections. We define the architecture as a pyramid, as each layer uses a subset of the nodes of the previous (below) layer.

In terms of functionality, the idea is to have a so-called SOTA module or process that operates independently and maintains an organized view of the network. By organization, we mean a subset of the network sorted based on an arbitrary metric. For example, as presented in the previous section, Kademlia organized the nodes into k-buckets based on their XOR distance with the local node identifier.

[1] `https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md`

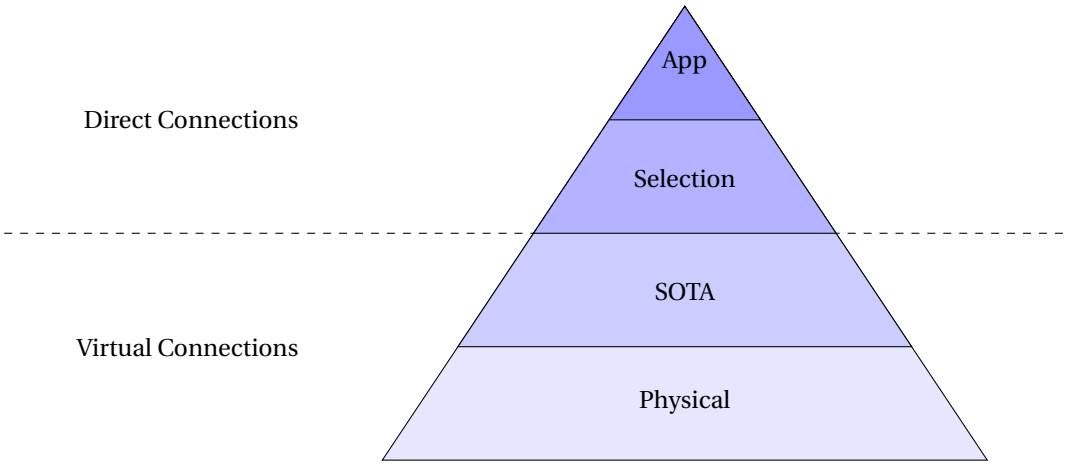


Figure 3.2: MPO Architecture.

Then, the selection layer would select nodes based on this organized set of nodes to build the final overlay the application will use. The selection is made with a customizable algorithm, such as selecting nodes in a round-robin manner across sorted groups, for instance. Moreover, when a direct connection stops, it is also up to the selection layer to repair the connection with a new node by still leveraging the organization provided by the SOTA module.

In addition to the SOTA module and the selection algorithm, the MPO protocol defines two other parameters: the requested number of **outbound connections** a node wants and the number of **inbound connections** it intends to allow. Similar connection types are used in the peer-to-peer communication of a Tendermint node [4]. Therefore, the selection will select the appropriate number of nodes based on the first parameter. Note that it might be the case that the outbound parameter is too high and the SOTA module does not provide enough peers.

We introduce the **extension module** to overcome this issue. This module is called when the node wants to discover more peers than the ones found by the SOTA layer. It initiates a discovery algorithm similar to the IPD algorithm presented in the previous chapter. This module is explained in more detail in the next section.

Moreover, every time a new peer is selected, the node must request it to be part of its connections. Since the places are limited on the peer, it might refuse the node. Therefore, we also refer to repairing the connection when encountering such a refusal.

3.2.2 Protocol

```
all_discovered_nodes = set<node>{}
outbound_nodes = set<node>{}
inbound_nodes = set<node>{}

connection_requests = timeCache<node>{}
```

Listing 3.1: Shared variables.

In the modules described below, we assume that they all share the above set of variables and that all can read and update them.

The variable `connection_requests` keeps track of the node to which the local node recently requested to have an outbound connection. This set is used to avoid asking the same node for a connection in a short period. We use a time cache because a node might refuse our connection but be available later. The time a node would remain in the cache heavily depends on the network usage and churn rate. Moreover, completely blocking nodes after a single connection refusal could lead the node to connection starvation in the future. Therefore, infinite blocking does not make sense.

Moreover, since multiple modules interact with the network, they can define callbacks when some network event occurs. We assume that all callbacks for a given event are executed sequentially in any order.

```
# Network event indications
event <newNodeConnected, node>
event <nodeDisconnected, node>
```

Listing 3.2: Network callbacks.

As presented with the architecture, both the SOTA and the extension modules will initiate contact with other nodes. Therefore, they will update the `all_discovered_nodes` set on their side using such network event callbacks. Moreover, note that it also means that a new node discovered by the extension module will be considered by the SOTA module, too, possibly updating its internal organization.

Despite being called a set for simplicity, note that we consider the `all_discovered_nodes` set to be capped. Indeed, otherwise, this set could increase infinitely. Therefore, consider it to contain the N most recently seen nodes and assume N to be bigger than the maximum number of nodes stored by the SOTA module and the requested number of outbound nodes so that we can ensure that selecting enough peers is theoretically possible.

We will reuse the request-response module initially defined in the previous chapter to build the new modules.

```
# Initialization
module RequestResponse<requestDataType, responseDataType>

# Requests
event <sendRequest, node, requestDataType>
event <sendResponse, node, responseDataType>

# Indications
event <receivedRequest, node, requestDataType>
event <receivedResponse, node, responseDataType>
event <noPendingRequest>
```

Listing 3.3: Request-response module interface.

We continue by defining the new **Extension** module with the following interface:

```
# Initialization
module Extension<>
uses:
    RequestResponse<set<node>, set<node>>

# Request
```

```
event <extend, target: int>

# Indication
event <done>
```

Listing 3.4: Extension module interface.

The role of the module is to find a target number of nodes using an internal algorithm similar to the one used in the IPD algorithm (i.e., the iterative fashion). Moreover, the nodes to find should satisfy the following conditions:

1. The node is not already outbound (i.e., part of the `outbound_nodes` set);
2. The node has not been recently requested (i.e., part of the `connection_requests` time cache).

In other words, we want to find nodes to which we can request a connection. When triggering the extension, the node we send the first request is chosen randomly among the `all_discovered_nodes` set. Moreover, if a new extension is requested while another is running, the module updates its internal target by adding to it the new target. Also, it is worth noting that the termination indication does not return the found peers; we assume they have been added to the `all_discovered_nodes` set, and, by the variable sharing, the SOTA module processed them. Moreover, it is also possible that there are not enough satisfying nodes in the network at a given time. Therefore, in this situation, the extension module will return when no more nodes are discovered. Also, note that the internal implementation leverages a time cache for nodes to which we requested their peers, similar to the time cache for connection requests. Finally, we consider the module to always be active in processing peer requests from other nodes, even when the main protocol does not request an extension.

We then define the **SOTA** module interface.

```
# Initialization
module SOTA<>

# Request
event <start, set<node>>

# Indication
event <bootstrapDone>

# Function
getOrganizedNodes() -> set<set<node>>
```

Listing 3.5: SOTA module interface.

It defines the start event to which we pass an initial set of nodes to allow the module to start bootstrapping. Then, once its internal bootstrap is done, the associated event is triggered. Note that this event is triggered only once! Moreover, the module provides a method to get an organized view of the nodes it discovered.

```
# Initialization
module Selector<>

# Function
selectNCandidates(int, set<set<node>>) -> set<node>
```

Listing 3.6: Selector module interface.

Finally, the **Selector** module interface above provides a single function for selecting the requested number of candidates. Moreover, note that this function might return fewer nodes than the amount requested, as it depends on the number of nodes provided.

With all the necessary modules defined, we now present the interface of the **Modular Pyramidal Overlay (MPO)** protocol.

```
# Initialization
module MPO<>(EXP_OUT: int, MAX_IN: int)
uses:
    SOTA<>
    Selector<>
    Extension<>
    RequestResponse<_, bool> as Connection

# Request
event <start, set<node>>
```

Listing 3.7: MPO module interface.

The protocol has parameters for the expected number of outbound connections and the maximum number of inbound connections we want. Moreover, it uses once again the request-response module that will be used to request nodes to be our outbound nodes. It also defines the start event to which we give the bootstrap nodes as a parameter.

Here is the final main algorithm of the MPO protocol:

```
# Node's local variable
state = IDLE

def send_connection_requests(nodes):
    for node in nodes:
        if not connection_requests.contains(node):
            trigger <Connection.sendRequest, node, _>
            connection_requests.insert(node)

def select_outbound_connections():
    n = EXP_OUT - len(outbound_nodes)
    org_nodes = SOTA.getOrganizedNodes()
    selected_nodes = Selector.selectNCandidates(n, org_nodes)
    outbound_nodes.insert_all(selected_nodes)
    send_connection_requests(selected_nodes)

def repair_outbound_connections():
    if len(inbound_nodes) > 0:
        node = inbound_nodes.pop_any()
        outbound_nodes.insert(node)
    else:
        org_nodes = SOTA.getOrganizedNodes()
        selected_nodes = Selector.selectNCandidates(1, org_nodes)
        if len(selected_nodes) != 1:
            state = EXTENDING
            trigger <Extension.extend, 1>
        else:
            outbound_nodes.insert_all(selected_nodes)
            send_connection_requests(selected_nodes)
```

```
30
31 upon event <start, bootstrap_nodes>:
32     state = BOOTSTRAP
33     trigger <SOTA.start, bootstrap_nodes>
34
35 upon event <SOTA.bootstrapDone>:
36     if len(all_discovered_nodes) < EXP_OUT:
37         state = EXTENDING
38         trigger <Extension.extend, EXP_OUT>
39     else:
40         state = IDLE
41         select_outbound_connections()
42
43 upon event <Extension.done>:
44     state = IDLE
45     select_outbound_connections()
46
47 upon event <Connection.receivedRequest, node, _>:
48     if outbound_nodes.contains(node):
49         trigger <Connection.sendResponse, node, true>
50     else if len(inbound_nodes) < MAX_IN:
51         inbound_nodes.insert(node)
52         trigger <Connection.sendResponse, node, true>
53     else:
54         trigger <Connection.sendResponse, node, false>
55
56 upon event <Connection.receivedResponse, node, accepted>:
57     if accepted:
58         # YAY!
59     else:
60         outbound_nodes.remove(node)
61         repair_outbound_connections()
62
63 upon event <Network.newNodeConnected, node>:
64     if state == IDLE and len(outbound_nodes) < EXP_OUT:
65         outbound_nodes.insert(node)
66         send_connection_requests([node])
67
68 upon event <Network.nodeDisconnected, node>:
69     if outbound_nodes.contains(node):
70         outbound_nodes.remove(node)
71         repair_outbound_connection()
72     else if inbound_nodes.contains(node):
73         inbound_nodes.remove(node)
```

Listing 3.8: MPO module algorithm.

The algorithm workflow goes as follows:

(a) The entry point is the start event (line 31), which directly initiates the bootstrap process of the SOTA module.

(b) When the bootstrap is done (line 35), did it find enough peers?
- Yes, the algorithm makes the first selection of outbound nodes (step c);
- No, the extension mechanism is triggered.

(c) The selection of the outbound nodes (line 10) is made using the SOTA and Selector modules, and a

connection request is sent to each candidate.

(d) When the extension is done (line 43), a selection is made (step c).

(e) When receiving a connection request (line 47), a node answers positively if the node requesting the connection is already outbound or if there is still someplace in the inbound nodes set (which, in that case, the node is added to).

(f) When receiving a connection response (line 56), a reparation (step g) is made if the response is negative.

(g) The reparation (line 17) first tries to upgrade one of its inbound connections if there are any; if not, the node asks the Selector module for a single candidate, and a connection request is sent to it. If no more candidates exist, the node initiates an extension to find more nodes.

(h) When a new node is connected (line 63), and the local node is in an IDLE state and is missing outbound connections, it will directly take the new node as a candidate and request a connection.

(i) Upon disconnection (line 68), a reparation (step g) is made if the node is an outbound connection.

3.2.3 Proof

To simplify the proof of the protocol properties, we assume the correctness of the sub-modules used:

- **Extension**: its correctness is inherited from the IPD algorithm as it follows the same behavior. Moreover, the search conditions do not impact the termination of the extension process, i.e., it will always eventually return.
- **SOTA**: its correctness followed one of the underlying state-of-the-art protocols used to maintain the organized view of the network. Moreover, the bootstrap process is also assumed to return eventually.

Moreover, we also assume that, among all nodes, the maximum value for the outbound parameter is strictly smaller than the minimum value for the inbound parameters. Otherwise, there is a risk that nodes will not find enough peers available to fulfill their outbound connections count. This is discussed in more detail in the discussion section 3.5 of this chapter.

the **scalability** property is inherited from the underlying protocol used in the SOTA layer. For the **self-organization** property, by the SOTA module's correctness assumption, the bootstrap protocol eventually terminates, triggering either an extension or the first selection of the outbound nodes. By the extension module's correctness assumption, it also eventually terminates and triggers the selection of outbound nodes. Therefore, the first selection is ensured.

After the selection, a certain number of connection requests is sent. All requests eventually get a response based on the correctness of the request-response module (proved in the previous chapter). Upon a negative response, an inbound connection is upgraded, another candidate is requested, or an extension is triggered if no more peers are available. Note that the algorithm cannot loop on reparation thanks to the time cache for connection requested nodes. Therefore, a node eventually gets all its outbound connections because of the earlier assumption about the outbound and inbound parameters. In the case of not enough peers in the network to satisfy the outbound parameter, the remaining missing connections will be initiated when new nodes join the network, ensuring the node will reach its desired number of outbound connections as soon as enough nodes are in the network. Finally, the self-organization property also holds when some connections stop due to the reparation triggering when the closed connection is outbound.

Moreover, there will never be more outbound connections than desired because events are processed sequen-

tially, and the candidate selection is only selecting the necessary amount. Furthermore, the fact that nodes are added to the outbound set before receiving the response ensures that multiple requests are not sent to a single place. This could lead to an overflow in the case of many positive responses. Regarding inbound connections, its cap is guaranteed by the size check before any insertion in the set. Therefore, the **degree control** property is ensured.

Finally, the **modularity** property holds due to the abstraction of the SOTA and Selector modules, allowing the usage of any organizational primitives.

3.3 Implementation

As for the IPD algorithm, the MPO protocol is implemented[2] using the rust-libp2p library too; please refer to the IPD implementation section 2.3 for a short introduction to the library.

The main difficulty in implementing the protocol is that all modules defined in the previous design section are *coexisting* in terms of event handling. In other words, we need to manually track the state of the protocol to know how to process a given event polled from the swarm. Here are the core components of the implementation in more detail:

- **Connection types**: As mentioned above, all modules use the same network stack. Therefore, despite the aim of restricting the number of connections with the selected nodes, the other modules still need to operate correctly. Indeed, a node might reach another for a simple request. However, with the library, a connection remains open unless explicitly closed. To counter this issue, we defined two types of connections: **ephemeral** and **persistent**. A connection to a node is marked as persistent if this node is either outbound or inbound. Otherwise, the connection is ephemeral. Moreover, an ephemeral connection is implemented using a timeout that automatically closes the connection when reached. Its value is not too short to avoid having a lot of *micro*-connections opening; it is in the order of a dozen seconds.
- **Bootstrap protocol**: The SOTA layer is defined as the bootstrap protocol in the code. It is implemented as a rust-libp2p behavior and enabled only if selected by the configuration. The behavior should have the functionality defined in the design section: a bootstrap protocol and a function returning the organized view of the discovered nodes.
- **Selector**: The selection is defined by a trait called `Selector` compromising a single function to select outbound candidates. The genericity brought by the usage of a Rust *trait* is a key point of the selection algorithm's flexibility.
- **Controller**: Previously called *Handler* in the implementation of the IPD algorithm, the controller is the core component tracking all dials, extension requests, connection requests, and the close channel (for ephemeral connections). Similar to the IPD algorithm implementation (section 2.3), each of these actions (except the connection closing one) has its corresponding configuration variables to define the concurrent factor and the maximum retries.

And so, the implementation uses three behaviors: **identify** and **request-response** provided by the library, and the custom **SOTA** one. To evaluate the MPO protocol, the SOTA protocol used is Kademlia, and we use the

[2] `https://github.com/informalsystems/malachite/tree/2625a833008a9ef53fd32f772778fe993f9e63a4/code/crates/discovery`

implementation[3] provided by the rust-libp2p library. Note that this implementation is slightly different from the original Kademlia paper. Indeed, the first difference is that the identifiers are 256 bits (using the sha256 hash function). The parameters k and α are set to 20 and 3 respectively. Moreover, the implementation is augmented with additional security concepts from S/Kademlia [2], such as parallel lookups over multiple disjoint paths.

3.4 Evaluation

3.4.1 Methodology

We evaluate the MPO protocol with the Kademlia protocol used as the state-of-the-art basis with the following selection algorithm: a round-robin across k-buckets starting from the furthest one (i.e., the largest k-bucket). For the outbound parameter, we set it to a fixed value of 20. We will vary the inbound parameter from 20 to infinity to evaluate its impact on the resulting network. We want to evaluate the following points:

- **Network distance distribution**: The global distance distribution in the resulting network.
- **Time**: The time it takes for a node to join a network to be able to define the overhead brought by the bootstrap of the SOTA module.
- **Kademlia balance**: The balance of the k-buckets among all nodes. More specifically, their size and the presence of each node in the remote k-buckets.

For the execution of the benchmark, we follow the same procedure as the join benchmark of the IPD algorithm: we start one node at a time and wait for the network to stabilize before spawning the next node (which has a single-node bootstrap set). Moreover, we go up to 500 nodes.

We use a worldwide distributed setup with Hetzner cloud machines (CCX43) with 16 dedicated vCPUs (AMD EPYC) and 64GB of RAM, deployed across Hillsboro (us-west), Nuremberg (eu-central), and Singapore (ap-southeast). Table 3.1 provides the round trip times (RTTs) between these regions.

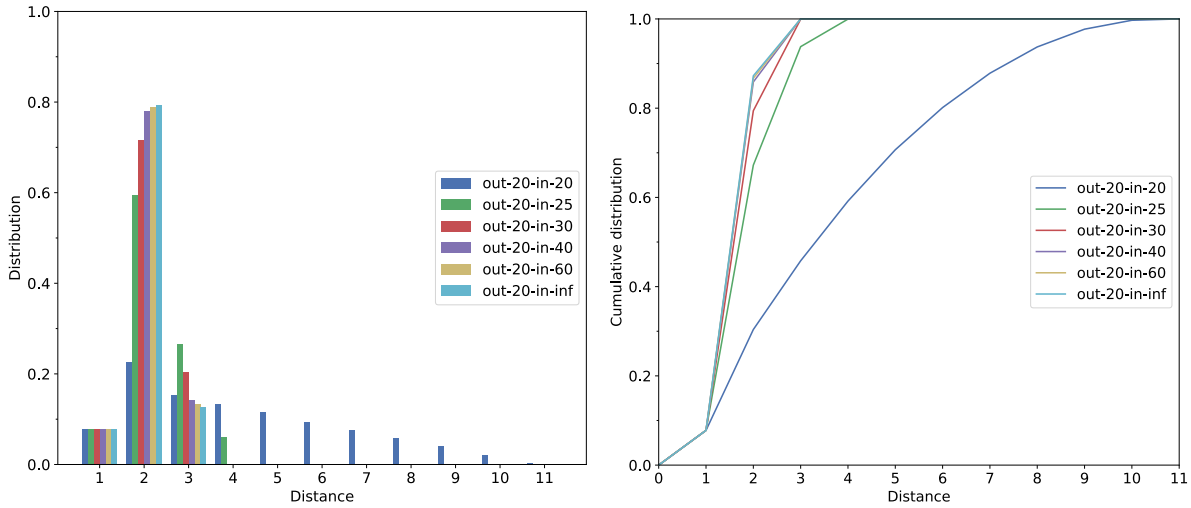
	us-west	**eu-central**	**ap-southeast**
us-west	0	173	187
eu-central	173	0	178
ap-southeast	187	178	0

Table 3.1: RTT (ms) between data centers.

3.4.2 Results

In Figure 3.3, we observe the impact of a small inbound parameter close to the outbound parameter. Indeed, a node takes the same amount of place it provides, leading to a network topology growing in length similar to the example network 2.9b, resulting in a maximum distance of 11. However, as soon as nodes provide more places than the one they are taking, the resulting network is better connected, and the maximum

[3] `https://docs.rs/libp2p/latest/libp2p/kad/index.html`



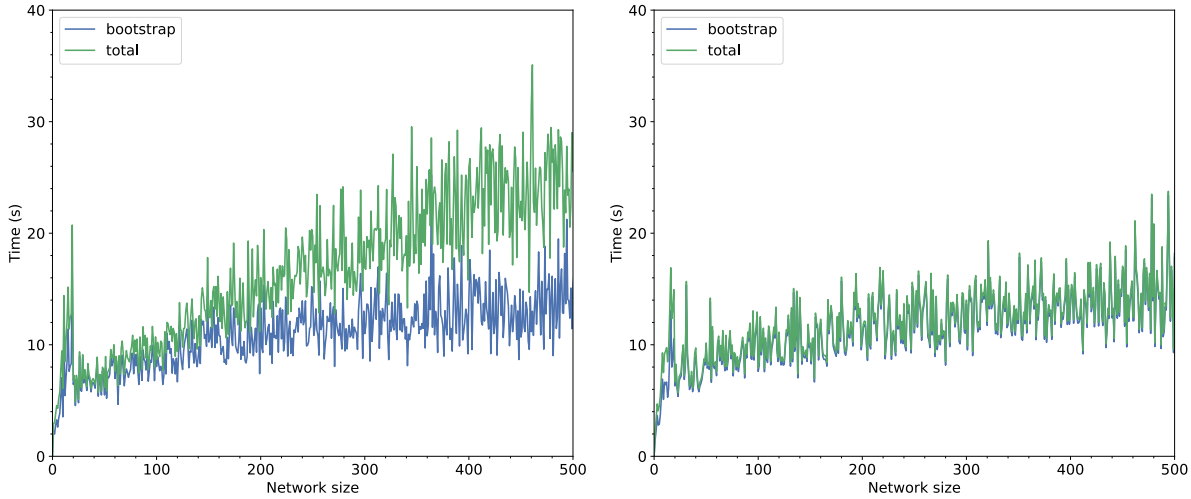
(a) Distribution.

(b) Cumulative distribution.

Figure 3.3: Network distance distribution.

distance is reduced to 4 with inbound = 25. Then, starting from the inbound parameter being twice the outbound parameter, the resulting network presents a reasonable distance distribution. Finally, observe that all configurations have a fixed number distance of 1 because the outbound parameter is the same among all runs, leading to the same amount of direct connections.

Also, despite having a different setup, our distribution is similar to the one obtained in the evaluation of the Tendermint protocol [4]. Their setup consisted of a 128-node network and outbound and inbound parameters, respectively, set to 10 and 40.



(a) Inbound = 20.

(b) Inbound = infinity.

Figure 3.4: Time to join a network with outbound = 20.

Regarding the time to join a network, we notice in Figure 3.4 that the bootstrap time does not depend on the

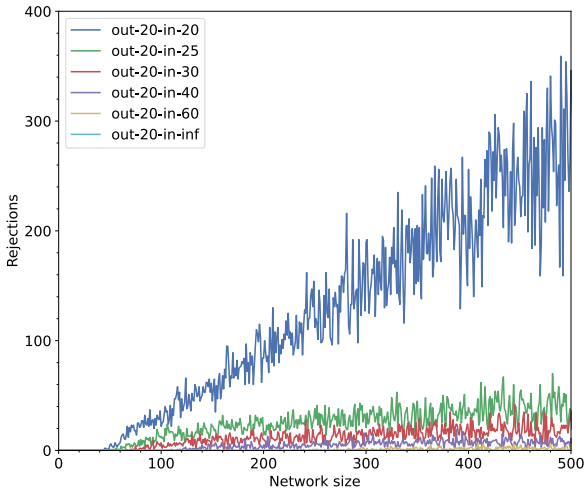


Figure 3.5: Rejections per node.

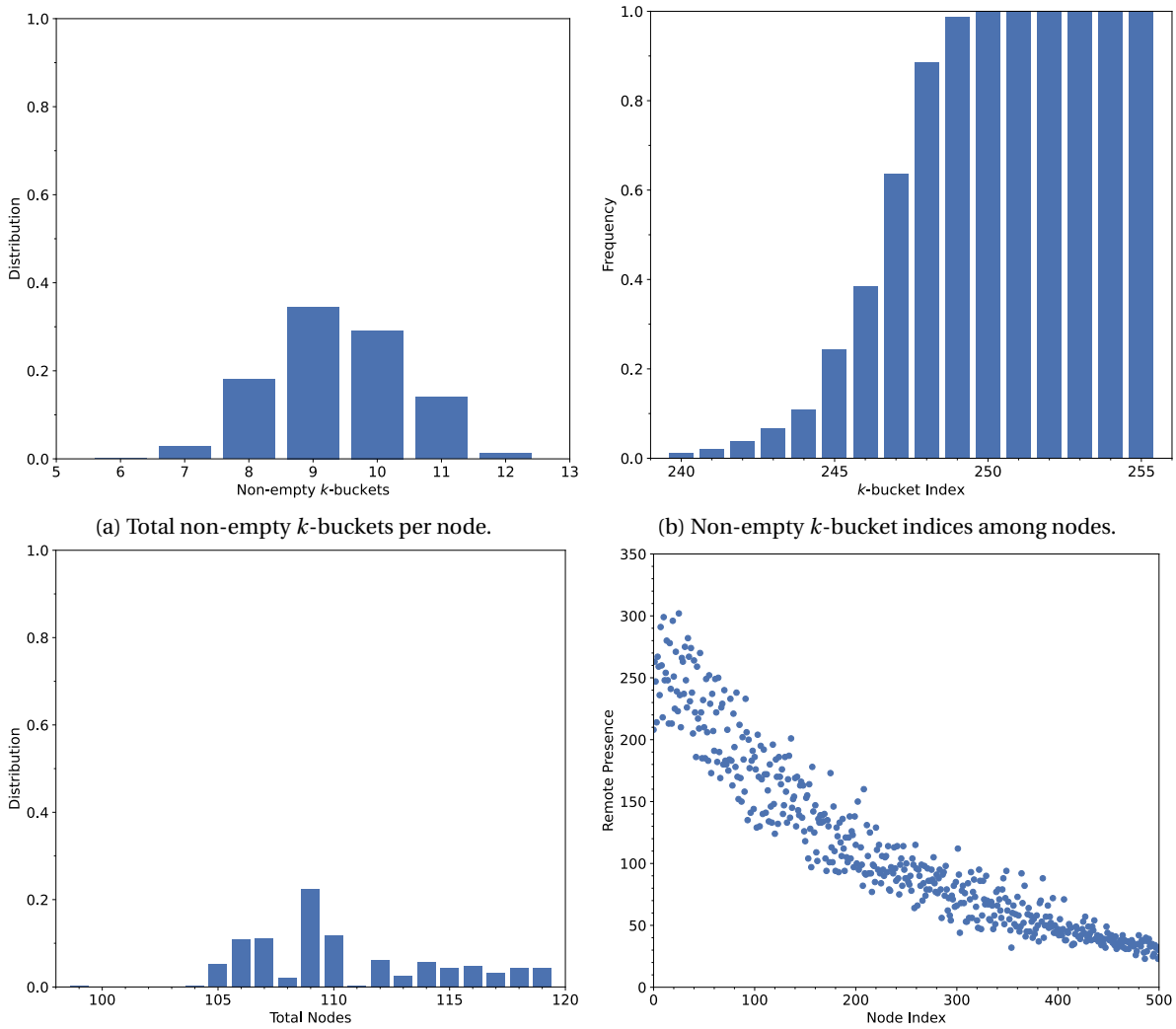
inbound parameter. However, the difference appears in the time needed to find the outbound connections. Indeed, as we previously mentioned, an inbound parameter of 20 leads to a constrained network, and the joining nodes are likely to send a connection request to an already full node; this behavior is showcased in Figure 3.5 with the number of connection rejections before finding their outbound nodes. Observe that the number of rejections is zero when the network has less than 20 nodes, as no nodes can be full yet. Back to Figure 3.4, we notice a peak for the first 20 nodes. This is due to the extension mechanism being triggered as the bootstrap did not find enough peers.

Regarding the k-buckets state, we first note in Figure 3.6a that nodes have between 6 and 12 non-empty k-buckets, with a median of 9. And if we have a closer look at which k-buckets are non-empty, we observe in Figure 3.6b that they are the biggest k-buckets in terms of distance with the local node. Indeed, it is worth noticing that theoretically, the first 10 k-buckets compromise more than 99.9% of the nodes in the key space, so it is unlikely that smaller k-buckets will be filled.

For the global balance, we first observe in Figure 3.6c that each node's k-buckets have a total of, in the median, 109 nodes. The most interesting result is that we observe a certain level of imbalance between the remote presence of nodes depending on their order of arrival (see Figure 3.6d). This result makes sense, as a node that joined the network earlier has a higher probability of getting discovered by the new node as it is already present in the k-buckets of most of the existing nodes. Moreover, nodes arriving later will not be able to insert themselves into other's k-buckets as these might already be full. Note that this was expected, as the implementation of Kademlia prioritizes older contact over new nodes, preventing k-buckets flush attacks.

3.5 Discussion

The core idea of the MPO protocol is the inheritance of properties from the underlying state-of-the-art protocol. Indeed, even if we reduce the number of connections, the background maintenance of the SOTA overlay supports the hold of these properties. However, it is worth noticing that this maintenance has a cost and a wrong choice in the underlying protocol could lead to high overhead. We observed it in the evaluation



(a) Total non-empty k-buckets per node.

(b) Non-empty k-bucket indices among nodes.

(c) Total nodes among k-buckets per node.

(d) Remote presence per node.

Figure 3.6: k-buckets state among nodes.

with Kademlia, where the nodes take at least 10-15 seconds to join relatively small networks (<1000 nodes) while, in comparison, it takes a few seconds for a simpler algorithm, as presented in the previous chapter. Therefore, choosing this protocol is crucial and reinforces the modularity capability of the protocol.

However, this modularity has a cost. Indeed, while generic solutions leverage a near *plug-and-play* experience for the engineers, our modular approach necessitates the implementation of the SOTA module. We can imagine that if such an approach becomes the way to go regarding overlays, we would get a kind of module library where it is likely that the specific organization metric one is looking for is already implemented. Using the MPO protocol would become relatively more manageable as one only needs to parametrize it to its will. This trade-off between genericity and performance applies to any topic, not just our case.

Another interesting point is the choice of the outbound and inbound parameters. Despite inheriting the properties of the SOTA protocol, if the resulting number of connections is small, there is a risk of being

disconnected during the short period required to repair the connections with the help of the SOTA overlay. Moreover, as quickly mentioned in the protocol's proof 3.2.3, we suppose that the chosen outbound parameter will be relatively small and quite constant throughout all nodes, and the inbound parameter will determine the connection adjustments. Indeed, to ensure a new node can find its requested number of outbound nodes, we theoretically need to ensure that the largest outbound parameter is smaller than the smallest inbound parameter among all nodes. It might sound like a consequent constraint, but we assume that, in a production network, nodes with a few resources (and so with low outbound and inbound parameters) are not the majority and are unlikely to become core nodes of the network. Therefore, even if the theoretical inequality is not respected, it does not make a network unable to operate correctly. Moreover, as mentioned in the background section 2.1 of the previous chapter, these resource-limited nodes can set themselves explicitly as not participating actively in the network.

Regarding related future work, it could involve a more comprehensive evaluation, particularly incorporating additional Byzantine and fault behaviors. This includes evaluating the resistance of the underlying SOTA overlay and the resulting network against high churn scenarios. In addition to resistance, one could also have a more precise evaluation of the overhead of the protocol. However, note that the performance of the underlying protocol mainly determines these evaluations.

Moreover, regarding the connections, a helpful feature would be to allow nodes to define *explicit* peers, that is, nodes to which they would be unconditionally connected. Implementing this feature would require careful consideration to ensure it does not interfere with the existing connection system. One possible approach could be introducing an additional connection type specifically for these explicit connections.

An interesting feature to explore would be to augment the MPO protocol with a topic or capability advertisement feature similar to the one implemented in the Discv5 protocol presented in the background section 3.1.3. Indeed, this feature would help find, for example, nodes with the validator capability in blockchain networks. One could also want to interact with so-called archive nodes, storing much of the blockchain history. More recent challenges are related to the decentralization of transaction sequencers[4]. A capability feature would allow nodes to efficiently route their transactions to the correct node responsible for the sequencing.

[4] `https://docs.starknet.io/architecture-and-concepts/network-architecture/starknet-architecture-overview/#sequencers`

Chapter 4

Evaluation of a Dissemination Algorithm

This chapter explores another critical aspect of distributed systems: information dissemination. Malachite currently employs the state-of-the-art GossipSub [24] for this purpose. Our objective is to evaluate a novel, alternative protocol, DOG [16], as a potential candidate for this role who aims to reduce the overall bandwidth consumption for message propagation. To address this, we conduct a comprehensive worldwide evaluation of these two protocols.

The chapter follows a similar structure as the previous ones; we start with a background section presenting the two protocols. We then present the details of the implementation of DOG. Next, the evaluation section explains the benchmark setup and presents the results. Finally, the chapter discusses potential improvements to the DOG protocol.

4.1 Background

4.1.1 GossipSub

GossipSub [24] is a gossip-based publish/subscribe (*pubsub*) protocol designed to achieve efficient, reliable, and attack-resilient message dissemination in open, permissionless blockchain environments such as Filecoin [14] and ETH2.0[1]. The primary objective is to propagate messages with minimal latency and bandwidth overhead in churn and BFT settings; nodes can join and leave at will, and some malicious actors might attempt to stall or disrupt the flow of information. GossipSub comprises two major components: the **mesh construction** and a **score function**.

When receiving a message, a node first checks whether it has already processed the message by consulting a **window-based** cache (called *mcache*) that stores recent message IDs. This mechanism prevents forwarding the same message multiple times, reducing redundant traffic and ensuring efficient use of the network resources. If the message is new, it is forwarded to specific nodes that are part of a so-called **local mesh**, which is a reciprocal overlay network where each node maintains direct links with a small number of peers, typically

[1] `https://blog.ethereum.org/2022/01/24/the-great-eth2-renaming`

between 6 and 12, referred to as the **mesh degree** or **amplification factor**. This **eager-push** mechanism ensures rapid local message propagation, allowing messages to spread quickly through tightly connected peers.

In parallel, the protocol employs a gossip layer for **lazy-pull** dissemination, which shared metadata (message IDs) of the messages in the *mcache* rather than the full messages themselves. This is done through `IHAVE` RPC messages, which notify a randomly selected fraction of the node's non-mesh peers (determined by the **gossip factor**) of the new message IDs. For example, with a factor of 0.25, the node selects 25% of its known non-mesh peers during each heartbeat (which occurs periodically, typically every second) to share message metadata. Upon receiving an `IHAVE` notification, people who discover they are missing a message can request the full message using an `IWANT` message. This approach enhances network-wide message reliability, ensuring that even nodes not directly connected via the mesh can retrieve missing messages through repeated gossip rounds.

It is important to note that mesh and gossip groups are not mutually exclusive; peers selected for gossip can also be part of the mesh. However, due to the rapid dissemination within the latter, these peers often have already received the message by the time the gossip round occurs.

A new message is initially published through **flood publishing**. The publisher sends the message to all (positively scored) peers who subscribed to the topic rather than confining it to the small mesh connections. Despite being more bandwidth-intensive for the publisher, this immediate flooding approach is highly effective at mitigating eclipse attacks since a publisher ensures that at least a portion of the network receives the message quickly, even if malicious actors attempt to isolate it. Afterward, the propagation continues with the usual mesh forwarding and metadata gossip.

In GossipSub, better connectivity in the mesh (i.e., maintaining a larger number of peers) reduces the propagation latency but increases bandwidth consumption, as more copies of the same message will be transmitted. Hence, there is a clear trade-off between message propagation speed and network resource usage. The protocol parameters (such as the amplification factor and the gossip factor) can be tuned by applications, allowing developers to balance low-latency message dissemination against the overhead of additional message traffic. Moreover, it is worth noticing that this trade-off is based on a similar reasoning as the one discussed in the limitation section 2.6 of the IPD algorithm.

A further important feature of GossipSub is the concept of **topics**, which provides a mechanism for message scoping. In practice, each node manages a separate mesh per topic, enabling fine-grained control over subscription preferences and network load. This structure helps limit the distribution of messages strictly to nodes that are interested in them, thereby reducing unnecessary traffic.

Alongside the mesh construction, the score function enforces reliability and security in adversarial conditions. Each node maintains a local score for every neighbor; these scores are never broadcast to other nodes, so a node makes routing decisions based on its own view of the network. Instead, each node applies its own weighted sum of performance indicators, with weights set according to the environment's requirements. For example, a node might assign positive weights to prompt message forwarding or long-standing, correct participation in the mesh and negative weights to invalid message propagation or suspicious behavior (such as dropping packets). Over time, nodes prune low-score peers from their meshes, graft new ones, and thus adaptively update the communication flow toward well-performing neighbors.

In the evaluation, we will use the rust-libp2p implementation[2] of GossipSub, which presents a few differences with the algorithm presented in the original paper:

- **Time cache**: The message IDs cache used is time-based instead of window-based. The default time-to-live for a message ID is 1 minute.
- **Score parameters**: The peer score system defines more parameters[3] than the ones presented in the paper.
- `IDONTWANT` **message**[4]: This message is broadcast when receiving a message bigger than a certain threshold. It aims to prevent the node's neighbors from sending it this large message, hence reducing the bandwidth usage.
- **Fanout**: In addition to the meshes, the node keeps a so-called fanout mapping topics to peers. More specifically, it only records topics the local node is not subscribed to, and the associated peers are selected to forward messages from these topics. Moreover, if the local node appears to join (i.e., subscribe to) the topic, it would simply create a mesh based on the fanout data of this topic, grafting each selected peer. Furthermore, when leaving (i.e., unsubscribing to) the topic, the mesh is dropped, and the peers are pruned. Finally, note that the fanout sets are discarded after inactivity for a while.

Plenty of other small details differ from the original paper, but we decided to stick to the most interesting ones.

4.1.2 Dynamic Optimal Graph (DOG)

The **Dynamic Optimal Graph (DOG)**[16] protocol is a gossip protocol developed to enhance CometBFT's memory pool (referred to as *mempool*) transactions dissemination. Initially, CometBFT employed a simple floodsub method, a naive gossip protocol where each node forwards newly received transactions to all its peers. While this ensures rapid propagation, it leads to excessive bandwidth consumption due to the proliferation of duplicate transactions. DOG addresses this inefficiency by minimizing redundant transmissions.

In protocols like GossipSub, mechanisms such as `IHAVE` and `IDONTWANT` effectively prevent unnecessary bandwidth usage by managing message-specific exchanges. DOG, however, introduces a more holistic approach by defining a metric called **redundancy**. This metric is calculated at regular intervals (typically every second) as the ratio of duplicate transactions to first-time transactions received by a node.

At the end of each interval, a node evaluates its redundancy level. If this level exceeds a predetermined upper bound, the node releases a `HaveTx` RPC message. This message is sent to the next peer from which a duplicate transaction is received. The dispatch of this control message carries the ID of the duplicate transaction. The sender, upon receipt, will then block the route from which it initially received the transaction to the node that sent the `HaveTx` message. This effectively prevents further traffic along that specific path. For instance, if node A sends a transaction to node B, which then forwards it to node C, and the latter identifies the transaction as a duplicate, it will send a `HaveTx` message to node B. Consequently, node B will cease forwarding transactions from node A to node C. It's important to note that within a single interval, only one `HaveTx` message can be sent, so only a single route can be blocked. The reason is to avoid a too-aggressive correction of redundancy.

[2] `https://docs.rs/libp2p/0.55.0/libp2p/gossipsub/index.html`
[3] `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md#overview-of-new-parameters`
[4] `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.2.md#idontwant-message`

Conversely, if the redundancy level falls below a specified lower bound, the node issues a `ResetRoute` message to reopen previously closed routes. This dynamic adjustment ensures that redundancy remains within an optimal range, typically around $1 \pm 10\%$, allowing for about one duplicate per transaction to maintain network resilience.

The transaction processing flow on a node is as follows: First, the transaction is added to the mempool, which records the transaction and its sender. This tracking is essential for effectively managing route closures. If the transaction is seen for the first time, it is forwarded to all nodes without a blocked route from the sender.

By design, DOG favors routes with lower latency. However, during its initial deployment, the network operates similarly to a floodsub system, requiring time to stabilize and achieve optimal efficiency. This initial phase is generally acceptable, as nodes typically remain active for extended periods, allowing the network to self-optimize over time.

4.2 Implementation

The only existing implementation of the DOG protocol is in CometBFT[5], where it is used for mempool transaction dissemination. This implementation is written in Go. In our case, we implemented DOG as a rust-libp2p protocol[6]. However, a few modifications were made compared to the CometBFT version:

- **Senders list simplification**: In the CometBFT implementation, the system operates with a high level of asynchrony. This necessitates maintaining a list of all nodes (ordered by arrival) from which a transaction was received. Since sending and receiving processes occur independently, there is a final check before forwarding a transaction to ensure the recipient did not originally send it. This mechanism prevents scenarios where two nodes simultaneously send the same transaction to the same peer, leading to unnecessary forwarding on both sides. However, rust-libp2p processes events sequentially, eliminating this issue[7]. As a result, it is sufficient to track only the first sender of a transaction rather than maintaining a complete list.
- **Optimized `ResetRoute` node selection**: In the CometBFT implementation, `ResetRoute` messages are currently sent to a random peer. This is suboptimal because the selected peer may never have received a `HaveTx` message from the node, meaning no previously blocked route will be reopened. Consequently, this delays the adjustment of redundancy until another interval when an appropriate peer receives a `ResetRoute` message. We optimize this selection by tracking each `HaveTx` message sent, ensuring coherence in the peer choice.
- **Time cache**: The CometBFT implementation is specifically designed for the mempool, meaning that transactions are removed from the cache (or mempool) when they are either included in a committed block or deemed invalid after a recheck. In contrast, our implementation employs a time-based cache, similar to the approach used in GossipSub. This adjustment is necessary since the protocol will not run alongside a consensus engine during the evaluation.

Furthermore, the DOG protocol was implemented at the same architectural level as the rust-libp2p GossipSub implementation, ensuring a consistent internal design. This common foundation allows for a fair and direct

[5] `https://github.com/cometbft/cometbft`

[6] `https://github.com/informalsystems/libp2p-dog`

[7] To be precise, the sending is also done asynchronously using sending queues, as in CometBFT. But, with the rust-libp2p architecture, it occurs at a level where the protocol state is not shared. We decided to stick to this clear processes separation design.

comparison between the two protocols under equivalent conditions.

4.3 Evaluation

4.3.1 Methodology

We evaluate both gossip protocols on two distinct network topologies: a fully connected network and a random network. Both configurations consist of 32 nodes. The random network is constructed using the MPO protocol described in the previous chapter, with Kademlia serving as the state-of-the-art (SOTA) and selection layers. The outbound and inbound connection parameters are set to 10 and 40, respectively.

Each experimental run lasts 20 minutes, during which all nodes publish 30 transactions per second, each with a size of 1kB. This results in a global workload of just under 1MB per second.

The GossipSub protocol is configured with its default parameters. Specifically, the mesh size parameters (`low`, `n`, and `high`) are set to 5, 6, and 12, respectively. The gossip factor is configured at 0.25. For DOG, the target redundancy is set to $1 \pm 10\%$, with a redundancy evaluation interval of 1 second. Finally, note that the message signature was disabled on both protocols.

We evaluate the following metrics:

- **Dissemination Time**: The time required for all published transactions or messages to propagate and reach every node in the network.
- **Bandwidth Consumption**: The primary metric of this evaluation, measured both globally across the entire network and individually for each node.
- **CPU Usage**: The computational overhead incurred by each protocol.

Additionally, for DOG, we measure the **redundancy** at each node to assess how quickly the protocol stabilizes.

Unlike the evaluations presented in earlier chapters, this experiment uses a simulated worldwide setup[8]. Synchronization is essential to accurately measure the publishing and delivery times of all transactions across machines. All machines are hosted within the same data center and synchronized using NTP with the closest available server to achieve this. The latency between machines is simulated using the Linux traffic control tool (*tc*), based on an arbitrary latency table (see Table 4.1). Moreover, the latency follows a normal distribution with a variation of ±5%. Each machine is assigned a simulated location in a round-robin manner.

For technical specifications, we use DigitalOcean[9] cloud machines with 4 dedicated vCPUs (Intel), 16GB of RAM, and network interfaces supporting up to 10Gbps bandwidth, all deployed in the Frankfurt (eu-central) data center. There is one machine per node.

[8] `https://github.com/informalsystems/libp2p-dog/tree/main/benchmark`
[9] `https://www.digitalocean.com/`

	N_Virginia	Canada	N_California	London	Oregon	Ireland	Frankfurt	S_Paulo	Tokyo	Mumbai	Sydney	Seoul	Singapore
N_Virginia	0	7	30	38	39	33	44	58	73	93	98	87	105
Canada	7	0	38	39	29	35	46	63	70	94	97	85	103
N_California	30	38	0	68	10	68	75	88	54	116	69	67	86
London	38	39	68	0	63	5	8	94	104	56	131	118	82
Oregon	39	29	10	63	0	59	68	88	49	109	69	63	84
Ireland	33	35	68	5	59	0	13	88	100	61	127	114	90
Frankfurt	44	46	75	8	68	13	0	101	111	60	143	109	77
S_Paulo	58	63	88	94	88	88	101	0	128	151	155	142	161
Tokyo	73	70	54	104	49	100	111	128	0	60	57	16	39
Mumbai	93	94	116	56	109	61	60	151	60	0	76	57	27
Sydney	98	97	69	131	69	127	143	155	57	76	0	69	45
Seoul	87	85	67	118	63	114	109	142	16	57	69	0	36
Singapore	105	103	86	82	84	90	77	161	39	27	45	36	0

Table 4.1: Latencies (ms) between worldwide data centers.

4.3.2 Results

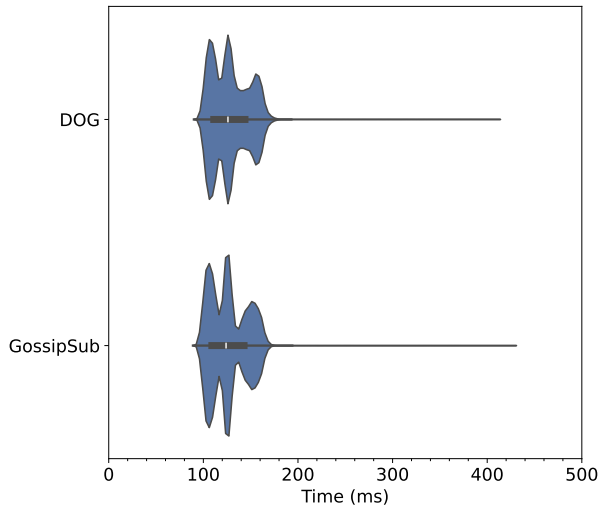
Figure 4.1 illustrates the distribution of dissemination times for both protocols. In the fully connected network (Figure 4.1a), both protocols exhibit similar performance, with a slight advantage for GossipSub, as reflected in Table 4.2. Both protocols display comparable peaks, whose presence likely follows the latency distribution of the experimental setup.

However, notable differences arise in the random network scenario (Figure 4.1b). GossipSub demonstrates two distinct performance zones, likely due to its dual dissemination mechanisms: eager-push and lazy-pull. The latter mechanism tends to slow down the overall dissemination process. Consequently, GossipSub's mean and median dissemination times are approximately 2.67 and 2.72 times as high as those of DOG, respectively (as shown in Table 4.2).

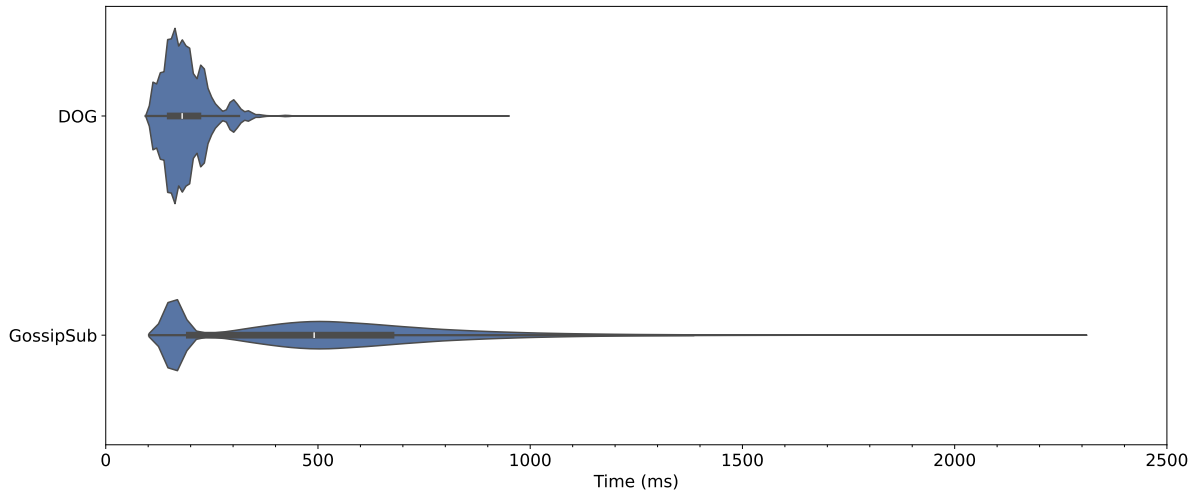
	Fully connected network		**Random network**	
	Mean	**Median**	**Mean**	**Median**
DOG	128	126	188	180
GossipSub	126	124	503	491

Table 4.2: Means and medians (ms) of dissemination times.

Figure 4.2 presents the overall bandwidth usage for both network configurations. As expected, DOG exhibits initially high bandwidth consumption due to its floodsub-like bootstrap phase. However, this usage steadily decreases as redundancy stabilizes (see Figure 4.4). In contrast, the mesh-based structure of GossipSub results in relatively stable bandwidth usage throughout the evaluation. Once stabilized, DOG demonstrates significant bandwidth savings. Detailed bandwidth consumption for individual nodes can be found in Figure 4.5 in the Appendix section of this chapter.



(a) Fully connected network.

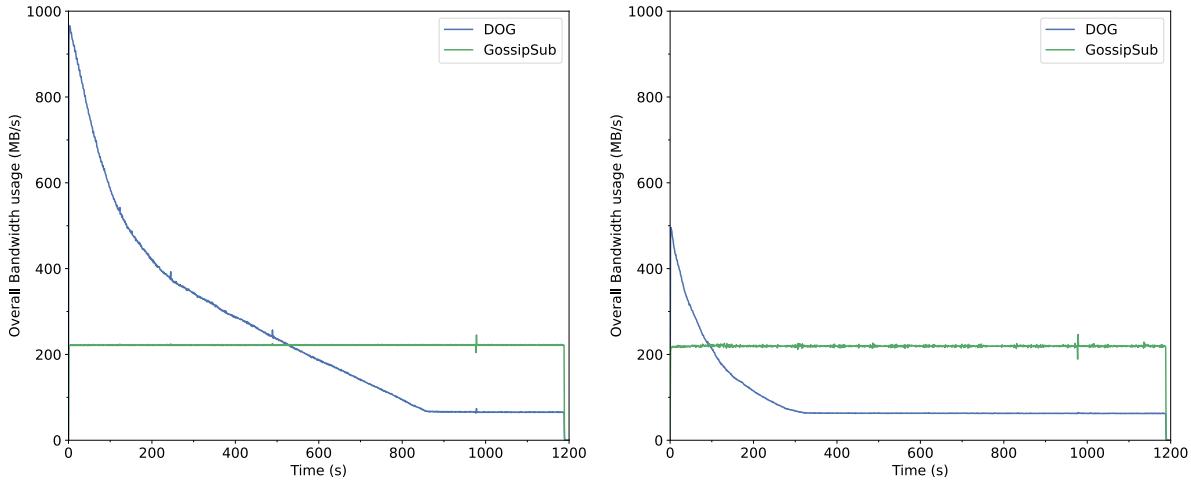


(b) Random network.

Figure 4.1: Distribution of dissemination times.

Regarding CPU usage, Figure 4.3 reveals a clear distinction between the two protocols. Despite its initial floodsub-like bootstrap phase, DOG consumes significantly less computational power. Across both network setups, GossipSub's CPU usage is approximately 11 times as high as that of DOG. This increased overhead can be partially attributed to GossipSub's peer scoring system, which DOG does not implement. Additionally, the higher CPU usage correlates with increased bandwidth consumption, as GossipSub handles a larger volume of messages. Detailed CPU usage per node is available in Figure 4.6 in the Appendix section of this chapter.

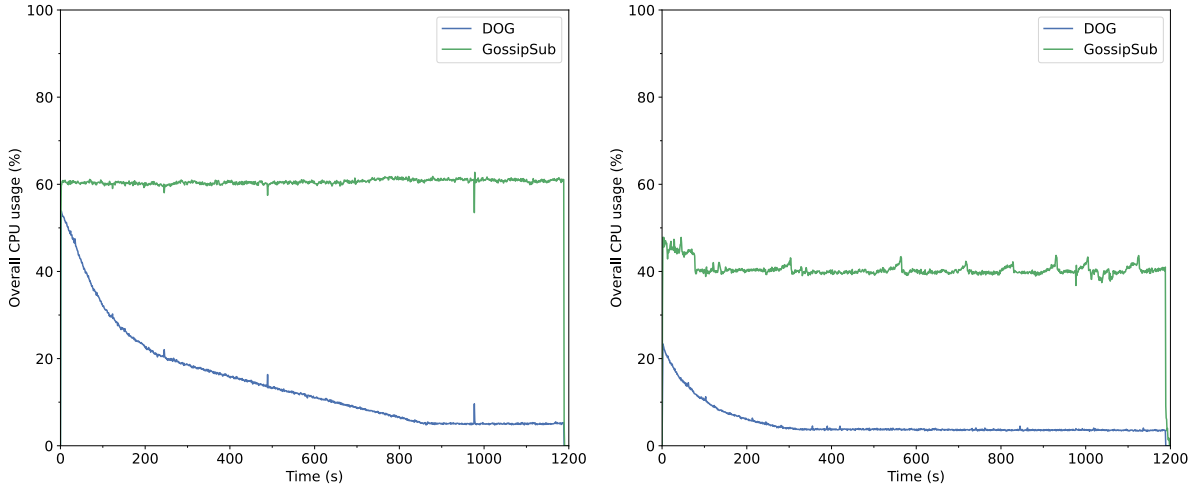
Finally, Figure 4.4 highlights the evolution of redundancy in DOG nodes. In the fully connected setup (Figure 4.4a), all nodes exhibit similar stabilization paths due to their identical initial connectivity. In contrast, the random network configuration results in varied stabilization times (Figure 4.4b), reflecting the non-uniform connectivity of nodes in this topology.



(a) Fully connected network.

(b) Random network.

Figure 4.2: Overall bandwidth usage.



(a) Fully connected network.

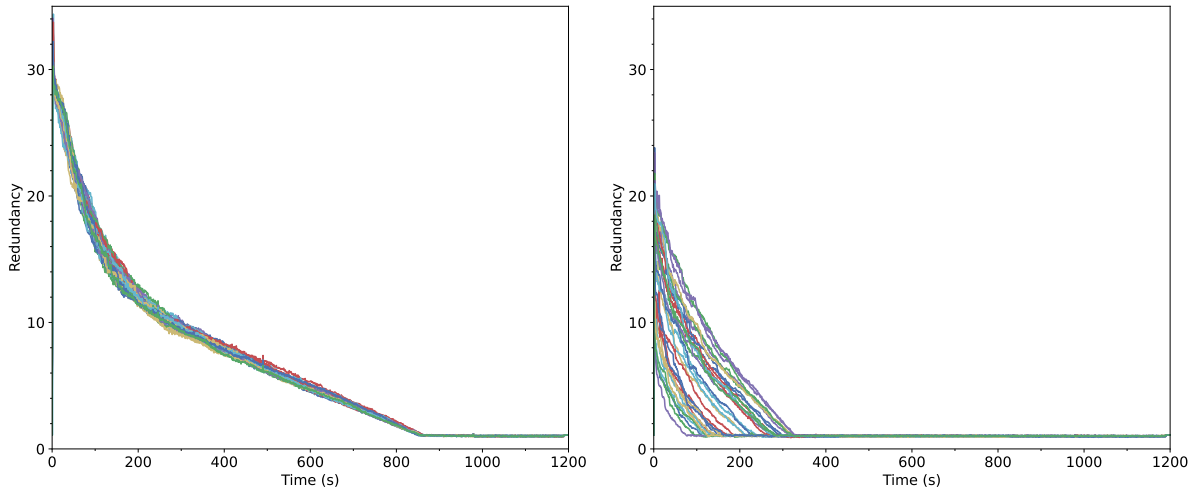
(b) Random network.

Figure 4.3: Overall CPU usage.

4.4 Discussion

DOG is a novel protocol with a lot of potential, as we saw in the evaluation against GossipSub. Moreover, it kept its promise of bandwidth saving. However, we did not evaluate the protocols against faulty or Byzantine scenarios, and a more extensive benchmark should be conducted to determine their resistance against such scenarios.

Moreover, there are different possible approaches to improve the performance of the DOG protocol even further. The rest of this section proposes such improvements.



(a) Fully connected network.

(b) Random network.

Figure 4.4: Detailed Redundancy.

4.4.1 Optimized Routing

In the current version of the algorithm, tracking transaction duplicates provides valuable information, particularly in determining the order of arrival for transactions. For instance, consider a scenario where no routes have been closed yet. If node X receives a transaction from nodes A, B, C, D, E, and F in that order, we can infer a high probability that future transactions following a similar network flow will arrive at node X in the same sequence. This establishes a *ranking* of speed, where earlier arrivals indicate faster routes. Consequently, assuming a redundancy of 2, closing routes from nodes D, E, and F would preserve the faster paths for transaction dissemination. Furthermore, if node A—the fastest route—crashes, it would be logical to unblock node D first, as it is likely faster than E and F.

Based on this observation, the algorithm could be refined by extending duplicate tracking to store the order of arrival for transactions. At the end of each redundancy interval, instead of only unblocking a single `HaveTx` message, we would leverage the recorded arrival order to close a slower route directly. This is feasible since a redundancy level exceeding the threshold guarantees that at least one transaction was relayed by more than redundancy + 1 nodes. Under the current algorithm, there is a risk of closing faster routes, as the `HaveTx` message is sent to the first duplicate sender detected in the next interval (which could be node B in our example). For the route unblocking, `HaveTx` message tracking (as discussed in Section 4.2) would be reordered based on the speed metric, ensuring that when unblocking is necessary, the fastest blocked route is prioritized.

This refinement would not directly impact the distribution time, as the fastest routes are unlikely to be mistakenly closed, even under the current algorithm. However, it would guarantee that if the fastest route becomes unavailable (e.g., due to a crash), the second-fastest route is already available (assuming a redundancy target set to at least 1) and that the fastest blocked route is prioritized for reopening.

Nonetheless, this enhancement assumes stable network latencies, which is not always realistic. Network topologies can change between the time a route is blocked and when it is later unblocked. However, this limitation primarily affects the `ResetRoute` optimization. The optimized `HaveTx` message handling remains valid even if the network topology changes, as it relies on data collected within the current interval—representing

the most recent state of the network. This optimization remains coherent since significant network restructuring is unlikely to occur between two consecutive intervals.

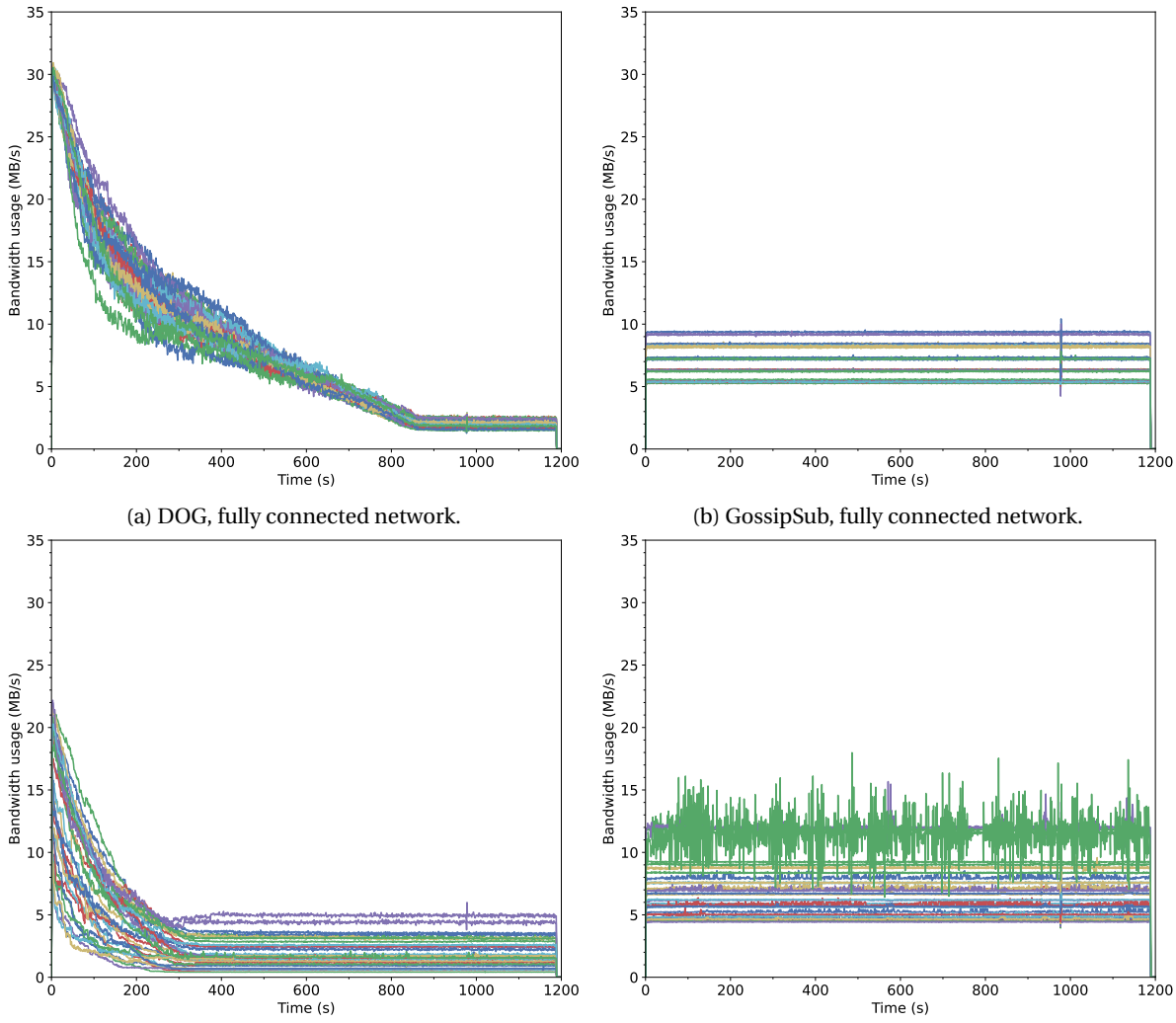
A potential method to optimize `ResetRoute` message handling could involve allowing blocked routes to compete for reopening. In this scenario, the node (that has a block route) that responds the fastest to a specific message would have its route unblocked first. However, this strategy would introduce additional complexity and computational overhead, which might outweigh any potential efficiency gains.

4.4.2 Stabilization Speed Enhancement

During the evaluation, we observed that DOG requires significant time to stabilize. In the fully connected network setup, the initial redundancy is approximately 30, given that there are 32 nodes in total. Since the protocol can only block one route per interval, the stabilization process progresses slowly, taking just under 15 minutes to reach the target redundancy of 1. To accelerate this adjustment, we propose allowing the protocol to send multiple `HaveTx` messages simultaneously when the current redundancy is significantly higher than the target. This concept is analogous to the learning rate in machine learning, where larger adjustments are made when further from the desired outcome.

However, it is crucial to determine the adjustment factor to prevent overly aggressive corrections carefully. Excessive route blocking could lead to network partitions, effectively isolating certain nodes and disrupting overall communication flow. To mitigate this risk, an empirical benchmark should be conducted to calibrate the adjustment rate based on the gap between current and target redundancy. This would ensure that the protocol adapts efficiently without compromising the stability or connectivity of the network.

4.5 Appendix



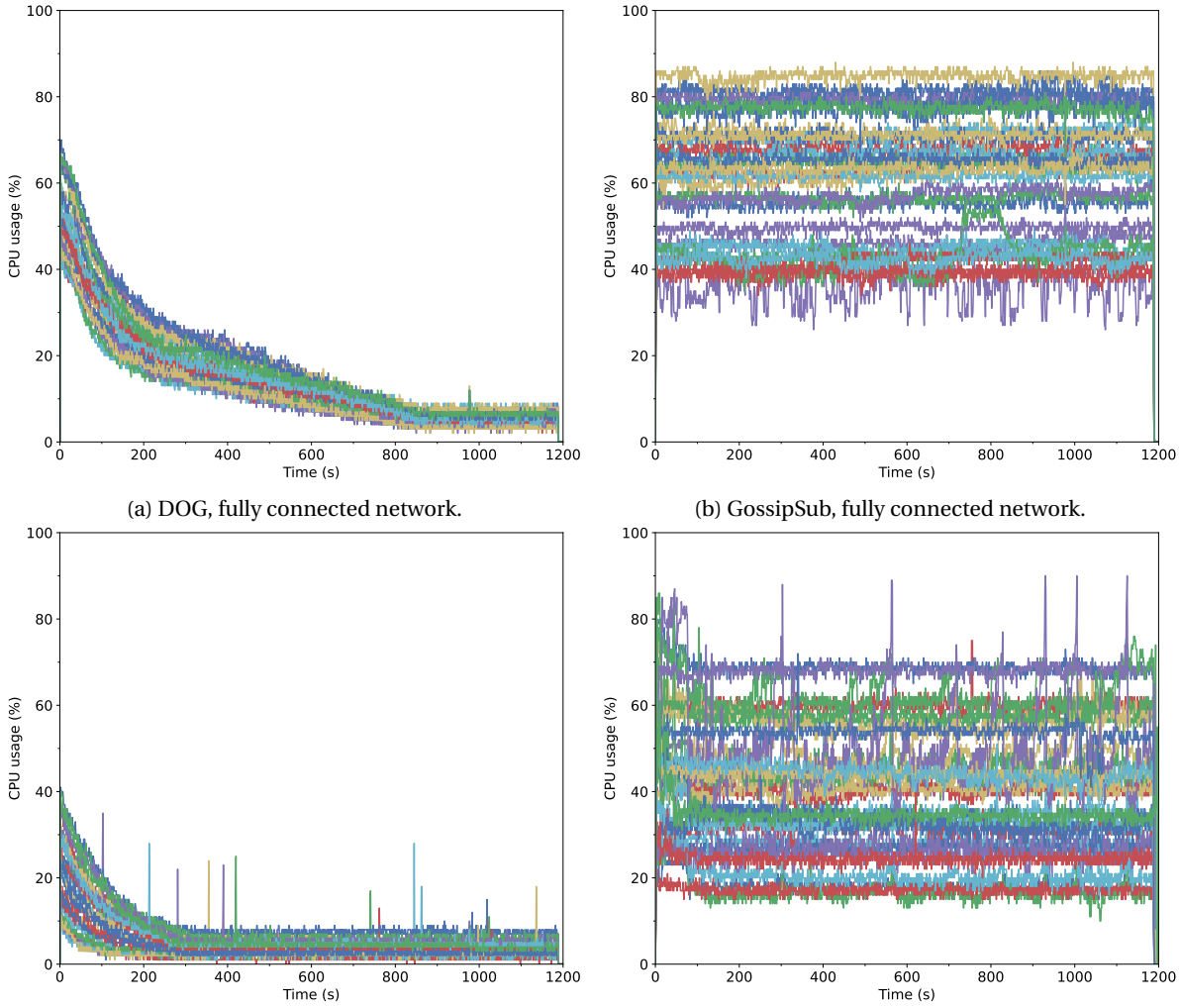
(a) DOG, fully connected network.

(b) GossipSub, fully connected network.

(c) DOG, random network.

(d) GossipSub, random network.

Figure 4.5: Detailed bandwidth usage.



(a) DOG, fully connected network.

(b) GossipSub, fully connected network.

(c) DOG, random network.

(d) GossipSub, random network.

Figure 4.6: Detailed CPU usage.

Chapter 5

Conclusion

In this thesis, we presented the following three contributions:

1. **An iterative approach to discovery**: We presented IPD, a simple and efficient node discovery algorithm. We successfully benchmarked it and showed that it allows a joining node to discover everyone in a worldwide network in a few seconds. Moreover, the algorithm showcased the capability to bootstrap a worldwide network from scratch in about 17 seconds. These results show that this algorithm is suitable for networks in the order of up to a few hundred nodes, leveraging simplicity and low overhead properties.
2. **A scalable and modular P2P network overlay protocol**: Despite being simple and efficient, the IPD algorithm presents a scalability issue. This led us to the world of network overlays, in which we proposed a modular approach through a pyramidal view of the overlay construction: the MPO protocol, compromising both discovery and overlay maintenance. This protocol leverages the Malachite philosophy of modularity, making it adaptable to any use case and allowing a fine-grained gestion of nodes' connections. We successfully showcased a proof of concept leveraging Kademlia, obtaining a balanced, resilient, and low-diameter network of 500 nodes.
3. **An evaluation of a novel gossip protocol**: We presented DOG, which mainly aims to reduce bandwidth consumption. We benchmarked it against GossipSub, a well-established gossip protocol, and successfully showed that DOG kept its promise. Indeed, it presented a significant saving in bandwidth. Moreover, its CPU consumption was nearly one-eleventh of the one of GossipSub. Then, we also presented potential improvements to the protocol to make it even more powerful.

With these contributions, we proposed and implemented novel, high-potential ways to solve two fundamental problems in distributed systems: node connectivity and information dissemination. More specifically, we presented a generic and modular approach to overlay maintenance, helping Malachite take one step further into its promise of decentralizing applications. Moreover, the DOG protocol showing such promising results reinforces the interest in digging more into its development to make it an even more performant and resilient protocol.

Bibliography

[1] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. "P-Grid: a self-organizing structured P2P system". In: *ACM SiGMOD Record* 32.3 (2003), pp. 29–33.

[2] Ingmar Baumgart and Sebastian Mies. "S/kademlia: A practicable approach towards secure key-based routing". In: *2007 International conference on parallel and distributed systems*. IEEE. 2007, pp. 1–8.

[3] Ethan Buchman, Jae Kwon, and Zarko Milosevic. "The latest gossip on BFT consensus". In: *arXiv preprint arXiv:1807.04938* (2018).

[4] Daniel Cason, Enrique Fynn, Nenad Milosevic, Zarko Milosevic, Ethan Buchman, and Fernando Pedone. "The design, architecture and performance of the tendermint blockchain network". In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2021, pp. 23–33.

[5] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S Wallach. "Secure routing for structured peer-to-peer overlay networks". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 299–314.

[6] Jochen Dinger and Oliver P Waldhorst. "Decentralized bootstrapping of P2P systems: A practical view". In: *NETWORKING 2009: 8th International IFIP-TC 6 Networking Conference, Aachen, Germany, May 11-15, 2009. Proceedings 8*. Springer. 2009, pp. 703–715.

[7] John R Douceur. "The sybil attack". In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 251–260.

[8] Robert A Ghanea-Hercock, Fang Wang, and Yaoru Sun. "Self-organizing and adaptive peer-to-peer network". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 36.6 (2006), pp. 1230–1236.

[9] Ken YK Hui, John CS Lui, and David KY Yau. "Small world overlay P2P networks". In: *Twelfth IEEE International Workshop on Quality of Service, 2004. IWQOS 2004*. IEEE. 2004, pp. 201–210.

[10] Hosagrahar V Jagadish, Beng Chin Ooi, Martin C Rinard, and Quang Hieu Vu. "Baton: A balanced tree structure for peer-to-peer networks". In: (2006).

[11] Márk Jelasity and Ozalp Babaoglu. "T-Man: Gossip-based overlay topology management". In: *International Workshop on Engineering Self-Organising Applications*. Springer. 2005, pp. 1–15.

[12] B Jhon and J Ari. "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks". In: *Proceedings of the Network and Distributed Systems Security Symposium*. 2012.

[13] Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. "The eigentrust algorithm for reputation management in p2p networks". In: *Proceedings of the 12th international conference on World Wide Web*. 2003, pp. 640–651.

[14] Protocol Labs. *Filecoin: A Decentralized Storage Network*. July 2017. URL: https://filecoin.io/filecoin.pdf.

[15] Lu Liu, Jie Xu, Duncan Russell, and Zongyang Luo. "Evolution of social models in peer-to-peer networking: Towards self-organising networks". In: *2009 Sixth International Conference on Fuzzy Systems and Knowledge Discovery*. Vol. 7. IEEE. 2009, pp. 250–254.

[16] Jasmina Malicevic, Sergio Mena, and Hernán Vanzetto. *Dynamic Optimal Graph (DOG) gossip protocol*. 2024. URL: https://github.com/cometbft/cometbft/blob/main/spec/mempool/gossip/dog.md.

[17] Petar Maymounkov and David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 53–65.

[18] Nick Rahimi, Koushik Sinha, Bidyut Gupta, Shahram Rahimi, and Narayan C Debnath. "LDEPTH: A low diameter hierarchical p2p network architecture". In: *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*. IEEE. 2016, pp. 832–837.

[19] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. "A scalable content-addressable network". In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. 2001, pp. 161–172.

[20] Matei Ripeanu and Ian Foster. "Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems". In: *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers 1*. Springer. 2002, pp. 85–93.

[21] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*. Springer. 2001, pp. 329–350.

[22] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM computer communication review* 31.4 (2001), pp. 149–160.

[23] Peter Triantafillou. "PLANES: The next step in peer-to-peer network architectures". In: *Proceedings of SIGCOMM Workshop on Future Directions in Network Architectures*. 2003.

[24] Dimitris Vyzovitis, Yusef Napora, Dirk McCormick, David Dias, and Yiannis Psaras. "Gossipsub: Attack-resilient message propagation in the filecoin and eth2. 0 networks". In: *arXiv preprint arXiv:2007.02754* (2020).

[25] Xiongfei Weng, Hongliang Yu, Guangyu Shi, Jian Chen, Xu Wang, Jing Sun, and Weimin Zheng. "Understanding locality-awareness in peer-to-peer systems". In: *2008 International Conference on Parallel Processing-Workshops*. IEEE. 2008, pp. 59–66.

[26] Rita H Wouhaybi and Andrew T Campbell. "Phenix: Supporting resilient low-diameter peer-to-peer topologies". In: *IEEE INFOCOM 2004*. Vol. 1. IEEE. 2004.

[27] Ben Yanbin Zhao, John Kubiatowicz, Anthony D Joseph, et al. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing". In: (2001).