A Practical Exploration of discv5 in Distributed Applications

Bachelor Thesis

to fulfill the requirements for the degree of
Bachelor of Science (B.Sc.)

at the
Hochschule für Technik und Wirtschaft Berlin

Faculty: Angewandte Informatik
Program: Applied Computer Science

First Examiner: Prof. Dr. Thomas Schwotzer
Second Examiner: Prof. Adrianna Alexander

Submitted by
Youssef Chammam
Submission Date: 11 August 2024

Abstract

This thesis explores the discv5 discovery protocol, analyzing its role and implementation in decentralized network systems. Beginning with an overview of distributed networks and their architectural overlays, the study delves into the operational principles of the Kademlia Distributed Hash Table (DHT) as a foundational technology for discv5. The evolution from discv4 to discv5 is scrutinized, focusing on the enhancements and their implications for decentralized networking.

The practical component of the thesis presents the integration of the discv5 Rust library into a prototype decentralized network. This network utilizes discv5 for node discovery and dynamic routing table management, coupled with a DHT for decentralized data storage, presenting a model for key-value storage in distributed environments. The implementation details are discussed, showcasing how the discv5 library can be adapted for real-world applications.

Extensive testing of the library under various scenarios—including IP changes, ENR updates, and concurrent requests—provides insights into its resilience and effectiveness against security threats, such as eclipse attacks. These tests highlight the library's capabilities and limitations within a simulated decentralized social media platform, demonstrating its potential to enhance privacy and reduce reliance on central authorities.

The findings underscore the viability of discv5 as a critical component for decentralized applications, emphasizing its security features, scalability, and adaptability. This thesis not only extends the theoretical understanding of distributed discovery protocols but also contributes practical insights into their application in enhancing decentralized digital ecosystems.

Contents

1 Introduction

1.1 Motivation

Decentralization has become increasingly popular over the course of the recent years due to their potential of distributing the power by eliminating the role of a central entity to govern a system. By distributing the governance of software across several nodes, this approach to data storage and management has recently gained in popularity and have proven their potential and benefits through the lack of a central authority and the high increase in privacy and freedom. However, designing such models require sophisticated and careful measures to make sure it complies with its nature while avoiding breaches to the whole system and remaining scalable and fast.

Over the last couple of months, X (also known as Twitter), has been taken over by Elon Musk in an attempt at "freedom of speech". His status and unconventional way of handling it, removing all rules, has made it increase in popularity drastically and is now considered the platform for freedom of speech by many, but it takes 5 minutes on the platform to see how it is nowhere near it. It is still the same centralized platform it used to be, except that it now has a new leader. Shadow-banning users, suppressing content and promoting other content is subjective to one's power over the platform.
The world is waking up to the need of a free platform, especially in unstable periods, and is coming to the realization that true freedom of speech is not a freedom that is decided by a single man, or a single community. The algorithm used is what governs the platform and rules it, and several attempts at decentralizing social media have risen. But the choice of technology used behind these platforms are not adequate for their purpose.

Recognizing how unethical X has become with no other real competitor that is making a real attempt in this mission, this project is an attempt at showing how the discv5 (discovery V5) technology can be used to fulfill this purpose. Discovery V5 is an extension to the Kademlia discovery protocol, with the intention of preventing attacks that exploit unverified endpoint information [23].
This bachelor thesis aims to explore the Rust library of the discv5 protocol, by understanding its functionalities on the networking level and demonstrating its application through the development of a simple, practical data storage, as well as testing the library under circumstances to represent how it reacts to given scenarios.
The research will assess the current landscape by investigating the existing frameworks and protocols underpinning distributed systems, with a particular emphasis on the role and significance of Distributed Hash Tables in enhancing privacy, security and distribution. It provides a comprehensive examination of the Discv5 protocol including its design and functionalities. Furthermore, it will bridge the gap between theoretical

understanding with a practical application offering insights into the challenges and opportunities of working with these technologies. Through this work, we will explore how networking protocols like discv5 can lead the path for the next generation of decentralized applications.

1.2 Project Objective & Proceeding

The objective of this bachelor thesis is to explore the Discv5 library practically as a mean to making a decentralized storage system for decentralized networks.
The primary objective being how the discv5 protocol can be integrated on the networking layer, exploring its functionalities and usability for networking purposes.
Once the discv5 is functional and well operating within our network, we will view how it can be integrated with another layer of key/value storage using Distributed Hash Tables across the nodes, for a fully decentralized network.
Finally, we will run discv5 through several tests to analyze and assess how it handles several scenarios that can occur between nodes participating in the network.
Meaning the final goal would be to have a deep exploration of discv5 theoretically, and to reinforce the theory learned by implementing it practically within an application to demonstrate its usability within a decentralized network. We will conclude with this paper on how it can serve as a laying foundation for networking over a decentralized network.

2 State of the Art

This section is dedicated to exploring key concepts and techniques that form the foundation of the node discovery V5 protocol. Understanding these concepts is crucial for developing and implementing an application based on discv5.
We begin by delving into the fundamentals of distributed systems in general, to assess where Distributed Hash Tables originate from and what problem they solve.
Next, we dive into the fundamentals of Distributed Hash Tables (DHTs), which is a distributed system that provides a lookup service similar to a hash table. Key–value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. [4]. DHTs, with their fast and reliable way of finding neighbouring nodes without the need of a central indexing node, are essential in building a decentralized application for data storage and retrieval. [4] This part will emphasize on Kademlia, an iteration over the concept of Distributed Hash Tables (DHTs). It is a specific type of DHT algorithm that was designed to improve the efficiency and performance of peer-to-peer networks by reducing the complexity of data look-ups. [30]
This part will finally cover the Discv5 protocol and how it implements the Kademlia algorithms solely for node discovery, excluding data retrieval. Specifically, it will have an in-depth coverage of its functionalities, while moving within its Discv5 library to comprehend the theoretical concepts proposed by it.

By comprehending these fundamental concepts, we can have a deep knowledge of how discv5 works and we lay the groundwork for developing a decentralized application based on this protocol.These concepts serve as the building blocks for the subsequent stages of our work.

2.1 Distributed Systems

2.1.1 Introduction to Distributed Systems

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." -Leslie Lamport [34] (page 4). This observation humorously yet effectively captures the interconnected and often complex nature of distributed systems, where components spread across multiple networked computers interact in ways that the failure of one can impact the functioning of others. These systems enhance performance, reliability and scalability by distributing tasks across multiple nodes, which can execute tasks simultaneously or sequentially.
Distributed systems are networks of independent computers that work together forming a single coherent system in order to achieve a common goal [33] (page 2). These systems leverage multiple computers, also called nodes to build a network of nodes that execute common tasks. A big range of possibilities are open, including sharing data across computers, communicating between them through packets, or emitting computational work results as events to one another, sharing information about the current state of the whole ecosystem. However, these advantages of distributed networks can have potential underlying issues that need to be handled with careful consideration, including more complex software, degradation of performance and often weaker security [33] (page 31). They are nevertheless at the core of decentralization, since with the help of architectures, when the nodes involved are not held by the same entity, migrate the system from having one central authority over the network (A), Figure 1 and data shared to a more cohesive system (B & C), Figure 1.

2.1.2 Characteristics

We can view a Decentralized Application as a sum of two important components: Data Storage and Node Discovery.
Both Discv5 and Kademlia DHT offer frameworks that can be tuned to meet these requirements. The realization of a distributed system requires that we develop a piece of software that is placed on the nodes across the network. But these nodes still need to find each other and to maintain a connection.Therefore, the architecture of the system depends on the underlying software that helps the nodes communicate as well.

Distributed Networking on the OSI Model

On the OSI (Open Systems Interconnection) Model, these protocols are situated in the 3rd Layer. The Network Layer, is the third layer in the OSI model. This layer is crucial

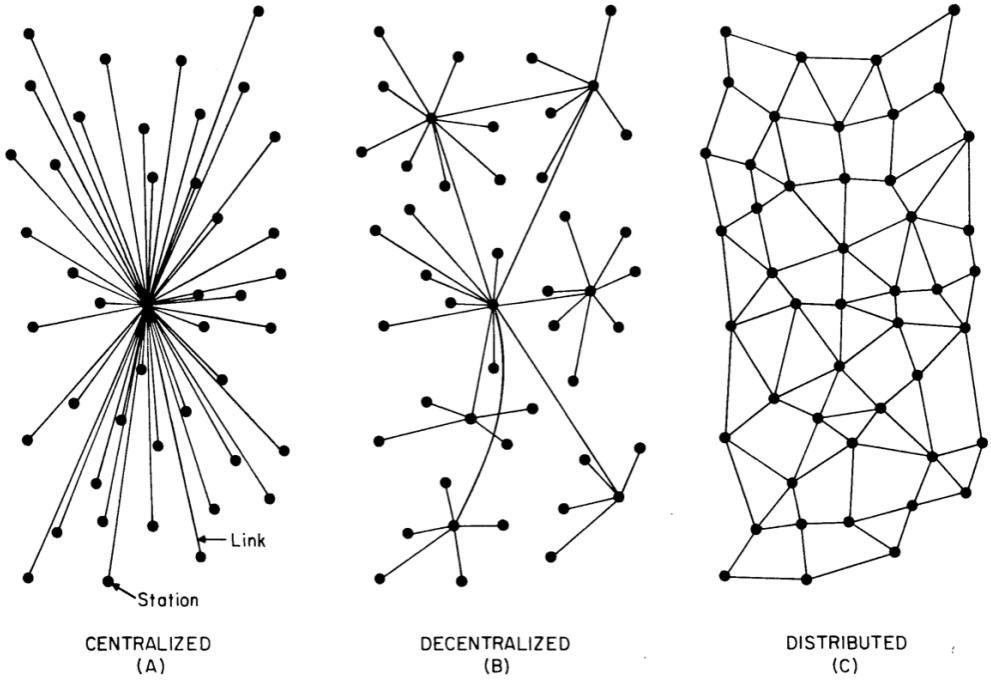


Figure 1: Centralized, Decentralized & Distributed Networks

for managing how data packets are sent and received across a network of computers. [13] The functions of the networking layer seen in figure 2 are:

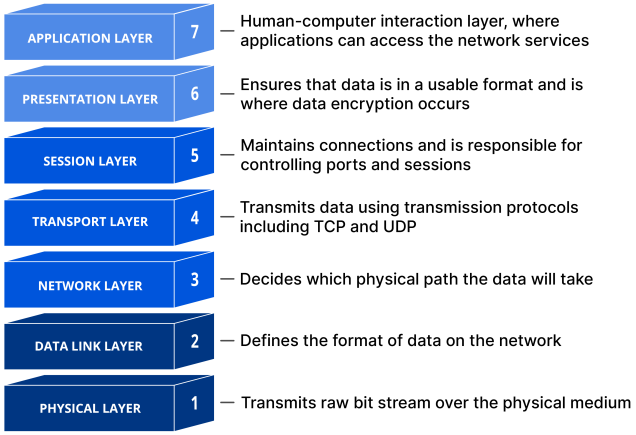


Figure 2: Layers of the Osi Model

- **Routing:** The Network Layer determines the physical or logical path that data

should take based on network conditions, connectivity, and the routing table.

- **Logical Addressing:** It assigns IP addresses to devices, which helps in identifying each device uniquely on the network.
- **Packet Handling:** The Network Layer handles the packets, including packet forwarding and packet switching, across various network segments and points.
- **Error Handling and Diagnostics:** It manages error reporting back to the source and network diagnostics.

2.1.3 Categories of Node Discovery in Distributed Systems

An important consideration is, that we are dealing with a collection of nodes within the system. Therefore, the software will need to manage and organize the nodes belonging to the system, allowing them to join and leave, all while maintaining a record of the nodes in the network. [34]
In practise, distributed systems are organized as an overlay network. Per definition, an overlay network is "a computer network that is layered on top of another (logical as opposed to physical) network." [6] A node is typically a software process equipped with a list of other processes it can directly send messages to. Messaging is sent in form of packets through TCP/IP protocol or UDP channels.
There are two types of overlay networks : *Unstructured overlay* In these overlays, each node has a number of references to randomly selected other nodes. This paper has a focus on the next type [33] (page 47). *Structured overlay* In this case, each node has a well-defined set of neighbors with whom it can communicate. The nodes are organized in a tree or logical ring. This is the case of distributed hash tables.
In both cases, an overlay network in general provides a very important principle: It should always be connected. Any node from the network can reach another node to pass a message, Which brings us to peer-to-peer (P2P) networking protocols [34] (page 4). Distributed systems can be classified into centralized or distributed computations (see figure 3). However for the sake of this study, we will focus solely on peer-to-peer networks. The p2p models can be divided into two categories : pure or hybrid. [35]
Pure p2p networks need to be carefully designed so that removing any single node chosen randomly from the network, does not have any impact on the whole system. However, in hybrid p2p networks, there are nodes that have some central authority over the network. Let's take the architecture of Napster, a renown peer to peer system for file sharing built in the 90s as an example to showcase the true benefits of pure p2p networks.
In hybrid p2p systems, there is a central server that maintains directories of information about registered nodes in the network, in the form of meta-data. This meta-data is a very important trait in distributed systems, as it allows the nodes to reach for a specific node to communicate with, and it can contain some information such as the IP-address, the Port as well as some characteristics about the node like the processing

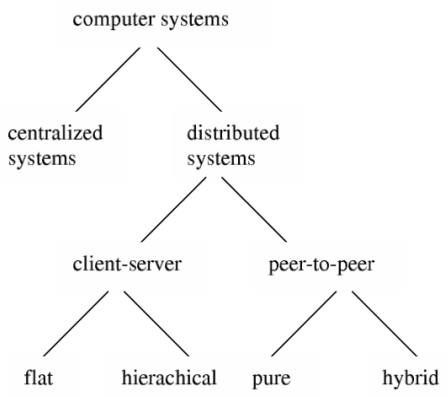


Figure 3: Computer systems

power, the free bandwidth, the location, the local timestamp of the process and in some cases some arbitrary data such as the additional services offered by that node. In the case of Napster [17], that demonstrates a centralized resource discovery architecture, the resource information about all other nodes are located at a central point that can be reached by any other node. The nodes periodically register their information to that central repository.
This approach has pros and cons. On the one hand, the reallocation of services can be more easily propagated across the network through a centralized discovery. On the other hand, we can say that this central authority (figure 4) presents power over the network, but what is more important is that an attack to this node is highly achievable, making it a vulnerability to the whole network, because without it, the network collapses [35] .

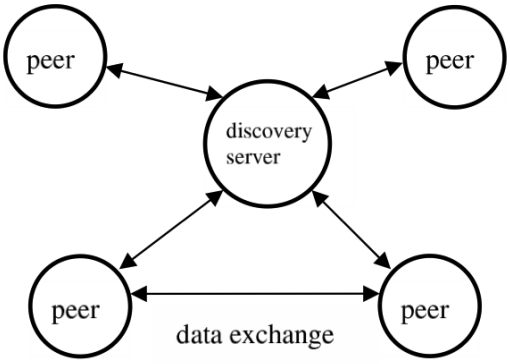


Figure 4: Centralized indexer

The thesis poses a challenge: developing a discovery protocol that adheres to decentralization while eliminating single points of failure and maintaining scalability and speed. A purely decentralized peer-to-peer (P2P) system lacks central control, with all nodes

functioning equivalently, contrasting the client-server model where a central node manages registrations. This structure enhances failure resistance and scalability.
P2P systems fall into two categories: unstructured and structured. Unstructured P2P systems lack a predetermined resource information distribution, making them robust against node failures and Denial of Service (DoS) attacks [14], but slower in resource discovery. Despite various efficient querying algorithms, such as the Random Pointer Jump Algorithm [14], the focus here is on structured systems due to their relevance to Distributed Hash Tables (DHTs), central to Discv5.
Structured P2P systems utilize a defined architecture, typically a tree (e.g., Kademlia) or a ring (e.g., Chord), facilitating rapid data querying and scalability. These systems employ unique identifiers for data storage and retrieval, organizing data into key-value pairs on each node.

Table 1: Comparison of P2P node discovery mechanisms

	Unstructured Pure p2p	**Structured Pure p2p**	**Centralized Indexing Hybrid p2p**	**Distributed Indexing Hybrid p2p**
Scalable	No	Yes	No	Yes
Flexible	Yes	No	No	Yes
Robustness	Yes	Yes	No	Yes
Manageable	No	Yes	Yes	Yes

2.2 Kademlia DHTs

2.2.1 Introduction to DHTs

Distributed hash tables, like stated in the previous chapter, is a structured peer-to-peer architecture. It is relevant, as it is the backbone of discv4 and discv5 [19].
It has come into existence around 2001 with the appearance of new protocols that propose a solution to plausible attacks to centralized systems like Napster, such as Tapestry, Chord, CAN and Pastry. All these protocols made having not a single one better than the other, but have several trade-offs. [30]
In DHT-based systems, we have a key-space. The key-space (typically 128-bit or 160-bit, 160-bit for Kademlia) [30] represents the total number of IDs that can be assigned within the system and it is opaque, meaning it never changes. In a DHT, data items receive an ID from this key-space and are stored within a key-value pair in a table, from which stems the name distributed hash table. What is interesting is, that nodes themselves receive a key from this same key-space. This table is distributed across the network of nodes. The key-value pairs of the data items are stored on nodes that have the closest ID to them. It is important to state that by "closest", it is not about geographical distance, however it refers to calculations based on the IDs. These calculations are made differently depending on the protocol, each following their own proposed

deterministic metric. Finally, a routing mechanism, which is what differentiates several DHTs from each other (Kademlia, Chord... are some examples of DHT-based information systems), is the routing mechanisms that the nodes will use to query some data by efficiently locating the node near any given target key, therefore the node storing the data.
Let's take a numeric example to better explain how this data is distributed: If we have a network with 12 key-value pairs and 4 nodes, each node would store 3 key-value pairs, including a key-value holding information about the node itself. This is of course an example. In reality, the distribution is almost always uneven, and the difference can be significant, depending on a given node's position within the key-space, and the distribution of keys. Any node can find any data needed within the system with its routing table by communicating with the neighbouring nodes.
For Kademlia nodes to support these functions, there are several messages with which the protocol communicates, also known as RPCs, or Remote Procedure Calls [19]:

- **PING:** The Ping RPC sends a ping to a given node to see if it is still alive.
- **STORE:** The Store RPC instructs a node to store a key, value pair.
- **FIND_NODE:** The FIND_NODE RPC takes only one argument, namely the 160-Bit ID of the node. The recipient of the RPC then returns the IP address, the UDP Port as well as the Node ID for the k nodes it knows about that are closest to the targeted ID.
- **FIND_VALUE:** Same as FIND_NODE, except if a node stores the specific value, it will return it immediately.

Node Lookup

The most important procedure from Kademlia is the *node lookup*. It is performed to locate the k closest nodes to some given node ID. During this process, an initiator node starts by querying ALPHA nodes from its closest non-empty k-bucket. The ALPHA parameter represents a concurrency limit set system-wide, typically three. This lookup process involves sending parallel, asynchronous FIND_NODE RPCs to these selected nodes.
As responses are received, if a node fails to respond promptly, it's excluded from future considerations. The initiator continually refines its list of k closest nodes based on new information received from responsive nodes, targeting subsequent queries to nodes progressively closer to the target ID. The process terminates once responses from the k closest nodes have been received, or if no closer nodes can be identified than those already queried. [30]

Network Participation and Bucket Population

To join the network, a new node must perform a lookup for its own node ID, thereby populating its k-buckets with the closest nodes found during this initial lookup. Fol-

lowing this, the new node inserts its contact information into the k-buckets of nodes it interacts with, ensuring that other nodes update their k-buckets to include the new node. [32]

2.2.2 Theoretical Foundations of Kademlia

As we delve deeper into the mechanics of distributed hash tables (DHTs), understanding Kademlia DHT is crucial due to its foundational role in earlier versions of node discovery, such as Node Discovery V4, which precedes Node Discovery V5 (discv5).
Kademlia primarily introduces a new mathematical approach to the "distance" calculations. Distance in quote because like stated earlier, it is not a physical distance, rather a metric distance based on mathematical computation between node IDs. [30]

2.2.3 Distance Calculation using XOR

Kademlia uses an XOR metric as its distance function for distance between points in the key space. XOR is a mathematical function that outputs true when inputs are different. In bits, when we combine 0 and 1, the output is true. And when we combine 0 and 0, or 1 and 1, the output is false.
XOR offers several characteristics that are beneficial for distributed systems as they reduce the querying complexity [19].

- **Zero Distance:** First off, the distance from one ID to itself results in 0, which is very practical for distance calculation between nodes. Let's say we have a DHT with 4 bits for the purpose of this example, and take a node with an ID of 1011. The distance between 1011 and 1011 (ones-self) is: First digit we have 1 to 1 which outputs 0 (false), second digit is again 1 to 1 (false), third is 0 to 0 (false) and finally 1 to 1 gives false as well. In the end we have a distance of 0.

- **Symmetry:** Another key characteristic of XOR is that distance is symmetric. Meaning that the distance from two theoretical nodes, node 1 to node 2, is the same distance as node 2 to node 1.

- **Triangle inequality:** Last but not least, the XOR metric follows triangle inequality. As an example with have a triangle with angles A, B and C. The distance from A to C is close or equal to that of A to B added to B to C.

With all these characteristics combined, the lookup algorithm of Kademlia results in having the ability of building a system where participants of the network are able to communicate with any other node within it by having the same distribution of nodes contained in their routing tables [16].
Figure 5 brief overview : It depicts on the one hand a theoretical global state of the distribution across the nodes (top part) , and on the other hand a representation of the routing table within a given node (bottom part).

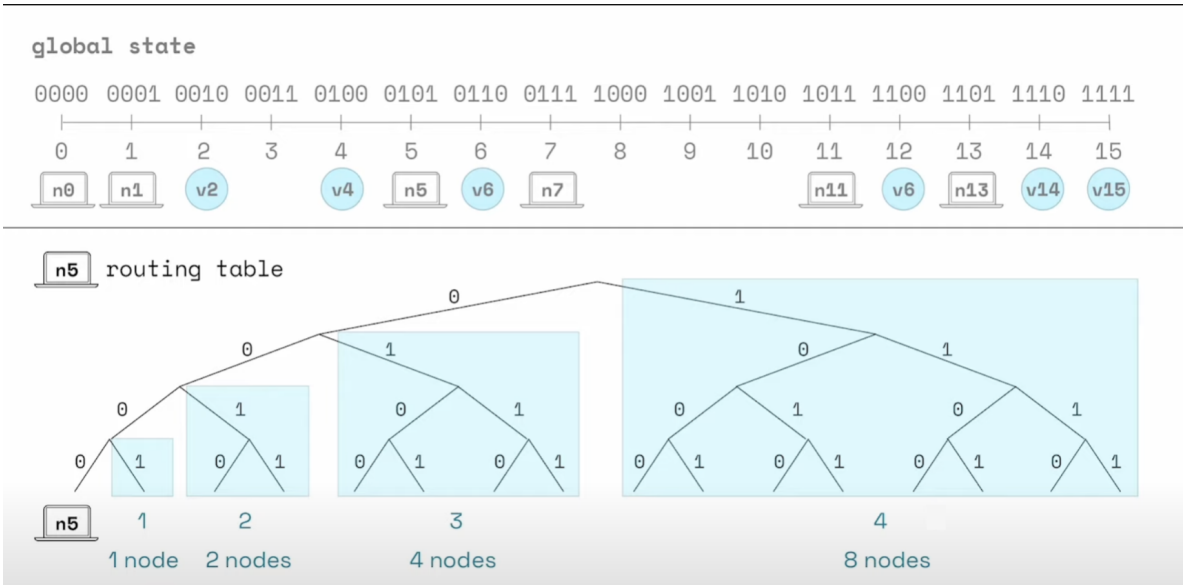


Figure 5: Kademlia binary tree. Nodes represented as laptops, data items as blue circles and k-bucket leaves in blue rectangles

2.2.4 Routing Table Structure

In Kademlia, the routing table of each subsequent node of the network is stored as a binary tree. [16] The structure of this tree is reflective of the binary representation of node identifiers within the network, where the total number of branches is determined by the number of bits in the system's identifiers. The leaves of this tree, or endpoints, correspond to the positions of nodes within the network. The spaces covered in blue rectangles in figure 5 can also be called buckets.

2.2.5 Bucket Management

Each node stores contact information about each other to route query messages. This information covers the IP address, the UDP port and the node ID. Upon every request, the node ID is passed along from one node to another to update the k-bucket for the sender's node ID. This node ID is utilized to reframe the bucket in which it is located. Upon reception, there are three different plausible scenarios that are handled in different ways:

- If it already exists within the specific bucket it belongs to, the recipient will typically move this node to the tail of the list. However in case the node ID does not exist in any bucket, the receiving node will simply insert it to the bucket it belongs to, if this bucket has less than k number of nodes registered. It does this by inserting it at the tail of the list as well.

- In the case the k-bucket is already full and has a k amount of nodes stored within

it, then the following occurs : It pings the least recently seen node within the bucket, which is stored as the head. If there is a response, the new node is discarded and the least recently seen node is now placed at the tail.

- In case the node does not respond to the ping, it is evicted from the bucket, and the sender node's id is placed at the tail. This way, all nodes stored within the buckets are always live. This practice is called in the official Kademlia paper a "least-recently seen eviction policy". [30]

2.2.6 Security and Efficiency in Kademlia

Although distributed network systems promise higher scalability, fault tolerance and performance when compared with traditional systems, there are also a few downsides that need to be managed carefully to determine that the system is secure and efficient. The most challenging among peer to peer systems in comparison with conventional client-server systems being 1.Security, 2.Reliability, 3.Flexibility and 4.load balancing. [35] Let's have a look on how Kademlia takes down these challenges.

Security

Numerous attacks can be made on Kademlia's key-based routing system as well as data storage, but for the purpose of this thesis, we will only cover the attacks on the routing system, because in the next chapter we will learn that data storage with Kademlia is not necessary within the Discv5 implementation. This section will summarize the possible attacks on the overlay routing.

- **Eclipse Attack :** This attack tries to place malicious nodes within the network, causing that one more more nodes are cut off from it by filling their routing tables, which will cause the messages to be routed to malicious nodes [15]. A node can be very subjective to this attack as soon as it has bootstrapped and come into creation, where there are probably way less nodes in its routing table.

- **Sybil Attack :** This attack can never be totally prevented within fully decentralized systems [15]. In decentralized network systems, there is no central authority that controls the delegation of node Ids a malicious node can take, which presents the possibility that a malicious node can take many, allowing him to control a space m from the limited key-space of IDs available.

Reliability & Load Balancing

In Kademlia, each bucket can have at best a specific number of nodes k that it cannot exceed, which is why they are called k-buckets.This size is set to balance the need for sufficient redundancy—ensuring that it is statistically unlikely for all nodes in a bucket to fail simultaneously within a short time frame, which defaults to 1 hour considering the Kademlia official paper [30]. This design choice enhances the fault tolerance of the

network, mitigating the impact of node failures on the network's overall functionality. This limitation on the number of nodes per bucket is a critical feature designed to ensure network robustness and efficient management of node contacts. Since we start with the node's position from left to right, with a limitation of k number of nodes per bucket, Kademlia is built in a way that nodes have more information about their neighboring. The arrangement of nodes within each k-bucket is also carefully managed: nodes are stored in a way that the "least recently seen" node is at the head of the list, while the "most recently contacted" node is at the tail. This method of organizing nodes within buckets ensures that more reliable and responsive nodes remain readily accessible, improving the efficiency of network queries and maintaining up-to-date information about the network's topology.

Scalability

Moreover, the architecture of Kademlia ensures that while each node maintains more detailed information about its immediate network neighbors (nodes in closer buckets), it retains the ability to connect to any part of the network. This is accomplished through the design principle that each k-bucket should ideally contain at least one node—if not full—thus enabling a node to reach any part of the system through a series of steps, contacting nodes progressively closer to the target part of the key space.
The duration of a search in Kademlia heavily depends on the correctness of a peer's routing table. The most crucial being its k closest neighbors in the overlay.[18] In the best case scenario, a peer should be able to return all of its k neighbors, in which case most operations take, following the big O notation, which is used to classify algorithms according to how their run time or space requirements grow as the input size grows [1], Olog(n) time to execute queries. [30]

This structure makes Kademlia particularly adept at handling large-scale, decentralized networks by optimizing both the speed of query resolution and the robustness against node failures, maintaining network integrity even as nodes continuously join and leave the network.

2.3 Discv5 Protocol

Having explored the intricacies of the Kademlia algorithm and its implementation within structured peer-to-peer systems, we now turn our attention to the evolution of these concepts within modern network protocols. In this chapter, we delve into discv5, the latest iteration of the Ethereum node discovery protocol. This protocol builds upon the foundational principles of Kademlia but introduces several enhancements aimed at improving security, scalability, and interoperability in increasingly complex network environments.

2.3.1 A brief overview of Discv4

Node Discovery V4, or discv4, is a kademlia-like DHT that stores information about nodes for networking purposes. It has been developed by the Ethereum foundation to implement it in Ethereum. Kademlia has been chosen mainly because of its efficient way at organizing a distributed index of nodes in a decentralized manner [27]. The discovery protocol itself is essentially based off of Kademlia, however, since it is purely for the node discovery, several aspects clarified in Kademlia are not regarded in it. Since it doesn't use the value portion from Kademlia, discv4 (or discv5) does not use **FIND_VALUE** as well as **STORE** RPCs [25].
Discv4 primarily uses Kademlia for the lookup method: A node contacts contacts some node and asks it for the nodes that are closest to itself, its neighboring nodes, in a recurring manner until it can no longer find new nodes.
It does so by adding extra information that is communicated upon contacting the nodes, which every node is expected to maintain. Namely, the Ethereum Node Record, or ENR for short. We will delve deeper in what the ENR is and how it is structured in section 2.3.2. The ENR can be requested by a new Remote Call Procedure introduced in discv4, by any node, which is called **ENR_REQUEST**.
Discv4 has been replaced by Discv5 because it included a few limitations:

- It is impossible to differenciate between node sub-protocols. In discv4, nodes were being added to the DHT without consideration of their communicating protocols. They were then eliminated upon contacting them, which is when it is determined that the node is invalid [19].

- Another limitation was the use of a node's local time clock to prevent Replay attacks, which is when data transmission is maliciously repeated or delayed. This attack involves the interception of a legitimate transaction that is then re-transmitted to produce an unauthorized effect by altering it. This has made a certain struggle in the Ethereum Network: When a machine's clock was off by a short timeframe, for example 2 minutes, it simply threw an error to reject the node from joining the network, making it inefficient [20].

- There was no way of telling whether or not both peers have verified each other. A node could consider another one verified, while the recipient did not verify the emitter node, forcing the recipient node to drop the FIND_NODE RPC [20].

All these issues makes Discv4 open for improvement.

2.3.2 Ethereum Node Records

The Discovery V5 Protocol uses Ethereum Node Records (ENR), to provide p2p connectivity information between the nodes within the Ethereum network. Therefore, understanding how ENR works is a must.

The motivation behind developing the Ethereum Node Records is because of the restrictions presented in the node discovery protocol: Node identity is discovered by knowing the node's identity public keys, their IP address and two port numbers. It is not possible to relay any extra information [28]. For that matter, the ENR proposes a more flexible format, *the node record*, for connectivity-related information, where it can hold not only the mentioned information (IP address, port...), but a record can also contain information about arbitrary transport protocols and public key material associated to them [22].

Why ?

Two keywords come in handy: **The Public Key Material** as well as **Arbitrary Transport Protocols**.
The **Public Key Material** within an ENR primarily pertains to cryptographic public keys associated with each node in the network. These keys serve several crucial purposes:

- **Identity Verification:** Each node in the network has a unique identifier derived from its public key. Nodes are therefore self-certified, and if a node's ID is known, the most recent version of its record can be retrieved.
- **Secure Communication:** Public keys enable nodes to establish encrypted communication channels, ensuring that data exchanged between nodes is secure from attacks such as eavesdropping and tampering.
- **Signatures:** ENRs are signed with the node's private key corresponding to the public key included in the ENR. This signature confirms the authenticity of the information in the ENR, such as IP address and port, so that receiving nodes can trust that the data has not been tampered with. This allows a reliable and secure peer-to-peer networking.

Arbitrary Transport Protocols: Transport protocols define the rules for data exchange over the network. By allowing ENRs to contain information about arbitrary transport protocols, Ethereum aims to achieve some goals towards a multi-chain network, the so-called Eth2.0 [19]. These include:

- **Flexibility and Extensibility / Interoperability:** Nodes can support multiple transport protocols simultaneously. This allows diversification within the same network ecosystem where different nodes can have varying properties and capabilities allowing them to practise different protocols within the same network.
- **Future-proofing / Protocol Upgrades:** As time moves on, new transport protocols can be developed, and integrated within the network with ease without the need of re-defining the discovery protocol. With ENRs, nodes can simply re-publish new ENRs that include these new protocols [22].

Specifications

A node record holds information that can be separated into 3: The signature, which is a cryptographic signature of record contents. The seq, which is a 64-bit unsigned integer representing the sequence number. This number should increase whenever the record is changed and re-published, so that contacting nodes are able to tell which record is the latest. And finally, the remainder consists of arbitrary key-value pairs [22].
The key/value pairs are sorted by their keys, and every key can only be present once. Key names have pre-defined meanings. We can find below a table of key value pairs that may be represented in an ENR. Only the id key is mandatory, and the rest are optional, even the endpoint, as long as the signature is valid.
Beyond the standard keys represented in the table, developers can define additional keys to store information relevant to their specific applications or sub-protocols. This allows for the inclusion of new features or support for additional protocols without altering the core ENR specification [22]. The records are represented as a Recursive-Length Prefix

Key	**Value**
id	name of identity scheme, e.g. "v4"
secp256k1	compressed secp256k1 public key, 33 bytes
ip	IPv4 address, 4 bytes
tcp	TCP port, big endian integer
udp	UDP port, big endian integer
ip6	IPv6 address, 16 bytes
tcp6	IPv6-specific TCP port, big endian integer
udp6	IPv6-specific UDP port, big endian integer

Table 2: Table of Key and Value pairs for ENR

(RLP) Serialization. It is a standard format for the transfer of data between nodes that is efficient. The RLP cannot exceed 300 bytes, and discovery implementations should reject any node with an RLP exceeding that specific number [29]. The limitaion is there because records are relayed frequently, and they may be included in size-constrained protocols like DNS [28].
Records are signed and encoded as follows:

```
content = [seq, k, v, ...]
signature = sign(content)
record = [signature, seq, k, v, ...]
```

Text Encoding of the RLP

Finally, the RLP can be then represented in a textual form, which is the base64 encoding of its RLP representation, prefixed by *enr:*.
To put everything into an example and wrap up the Ethereum Node record, we have the following example from the official EIP proposal [28]: Information that is encoded:

- node ID:
  a448f24c6d18e575453db13171562b71999873db5b286df957af199ec94617f7
- IP address: 127.0.0.1
- port: 30303

The record is signed using the "v4" scheme using the **seq** (sequence number) 1 and the private key : b71c71a67e1177ad4e901695e1b4b9ee17ae16c6668d313eac2f96dbcda3f291

The RLP structure of the record is :

```
[
  7098ad865b00a582051940cb9cf36836572411a4727878...76f2635f4e234738f30...,
  01,
  "id",
  "v4",
  "ip",
  7f000001,
  "secp256k1",
  03ca634cae0d49acb401d8a4c6b6fe8c55b70d115bf400769cc1400f3258cd3138,
  "udp",
  765f,
]
```

The Text encoding of the RLP is (dotted to shorten):

```
enr:-IS4QHCYrYZbA..HcBFZntXNFrdvJjX04jRzjzCBO..._oxVtw0RW_QAdpzBQA8
```

2.3.3 Introduction to Discv5

Discv5 is a standalone protocol that operates on UDP over a dedicated port, designed to address the shortcomings of its predecessor, Discv4. This protocol introduces significant enhancements, primarily through the implementation of flexible peer records known as Ethereum Node Records (ENRs). The rationale behind Discv5 is to achieve a series of general and security-specific goals, enhancing reliability, security, and future scalability of node discovery mechanisms. [23]

Why Discv5 ?

According to the Ethereum Foundation [21], libp2p Kademlia DHT is a fully-fledged DHT protocol/implementation with content routing and storage capabilities, both of which are irrelevant in the context of Peer Discovery.

2.3.4 Wire Protocol of Discv5

Having discussed Ethereum Node Records (ENRs), which are pivotal in providing comprehensive and verifiable node information within the Ethereum network, we now transition to another critical component of Discv5: the wire protocol. ENRs serve as the backbone for node identity and metadata [28], but the practical application of these records is realized through the wire protocol, which handles the networking between nodes [26].
The wire protocol in Discv5 is the set of technical standards and procedures that dictate how nodes can exchange information. This protocol is crucial because it encapsulates the methods for initiating and maintaining communication, ensuring data integrity, and providing security through encryption [26]. The wire protocol's design reflects an evolved understanding of network challenges and security threats, building on the shortcomings of previous versions [19].

UDP Communication

Node discovery messages are sent as UDP datagrams [26]. As described in an example distributed System on the OSI Model in figure 6, the Wire protocol from the discv5 operating at the networking level (3), makes use of UDP for communication in the Transport Layer (4).
The wire protocol mandates the use of UDP for no one big reason, but several smaller ones. These reasons come in 5 parts [31]:

- NAT Traversal: UDP is chosen for its simplicity in handling NAT (Network Address Translation) issues. Unlike TCP, UDP doesn't require complex setup processes for NAT traversal. UDP allows packets from any other node to reach the node behind the NAT without additional configuation, allowing more reliable communication across the system even with different network setups, unlike TCP.

- Uniform implementation Across Nodes: UDP ensures that all nodes implement the same protocol of communication. This uniformity is crucial for discv5 because all nodes must be able to communicate with each other without any compatibility issues.

- Reduced Communication Latency: UDP allows for faster communication, which is necessary in node discovery and maintenance. The protocol requires nodes to quickly interact with many others to update neighbor lists and verify the availability of other nodes. Also, as we will uncover shortly, packets are split into several ones because of size constraints. And UDP doesn't require the receiver to actually confirm the reception of packets, unlike TCP.

- Handling of High Connection Numbers: Using UDP avoids the overhead associated with managing numerous TCP connections. If TCP were used, nodes might either run out of system resources like file descriptors due to the need to maintain

connections with many nodes, or they would have to constantly establish and terminate TCP connections. UDP, by not requiring permanent connections, saves resources and simplified communication processes.

- Resilience to Packet Loss: UDP's tolerance for packet loss enhances the robustness of the protocol. In UDP, it is acceptable for packets to be lost and not reach their destination. This feature is beneficial as it forces the protocol to be resilient even in poor network conditions, allowing implementations to intentionally drop packets to manage traffic and computational loads without compromising the network's functionality. Another strategic advantage is the fact it helps masking whether non-responses are due to intentional decisions or network issues, adding a layer of unpredictability.

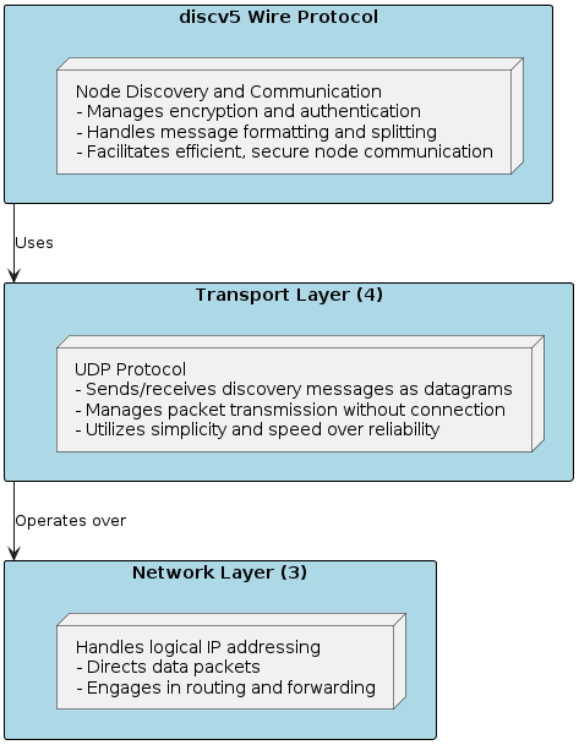


Figure 6: Integration of the Discv5 Wire Protocol with the OSI Model

The Wire Protocol specifies that the maximum size of any packet is 1280 bytes; any implementation should not generate packets larger than this size. And the minimum packet should be 63 bytes. Packets lower than this size should be rejected. [26] In order to stay between these sizes, packets that exceed the upper-limit should be split into multiple smaller ones and send multiple messages all while specifying the total number of responses in the message. [26]
Since the protocol expects low-latency communication, there is a short timeout of 500ms

for single request/response interactions (RPC interactions), and a 1 second limitation for handshakes. Furthermore, when responding to a request, the response should be sent to the UDP envelope address of the request. [26]

Packet Encoding

There are 3 distinct types of packets in the wire protocol, that are all 3 tied to each other in their logic [31]:

- **Ordinary message packets**: Packets carrying an encrypted/authenticated message
- **WHOAREYOU packets**: These packets are sent to a requester when a recipient of an ordinary message packet cannot decrypt or authenticate the packet's message
- **Handshake message packets**: These packets are always sent following the WHOAREYOU packets. They carry handshake-related data in addition to the encrypted/authenticated message first send by the ordinary message packet followed by the WHOAREYOU packet.

Protocol Header Overview

In Discv5, every discovery packet starts with a header, which is encrypted to prevent firewall detection. This header is constructed as follows:

- **Masking:** The header information, including the protocol ID, version, and message metadata, is masked using AES/CTR encryption with a key derived from the destination node ID and a packet-specific random IV.

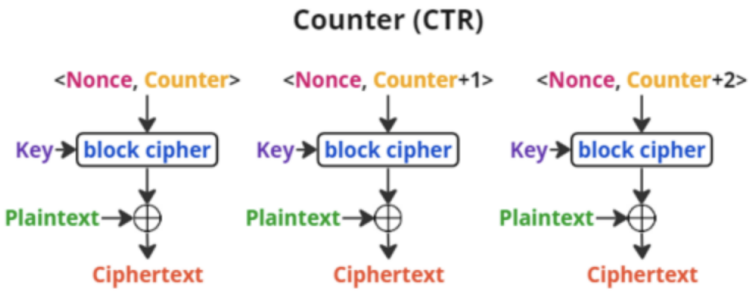


Figure 7: AES/CTR Encryption

AES is a widely used encryption standard that converts plain text into unreadable text (ciphertext) using a secret key. CTR (Counter) Mode turns AES into a stream cipher, which means it can encrypt data of any size, byte by byte. It does

this by combining the encryption of a counter value (the nonce) with the plain text to produce the ciphertext [2].

- **Structure:** The actual header consists of a static header and an authdata section. The static header includes the protocol ID ("discv5"), version (0x0001), packet type (flag), and a nonce. A nonce is an arbitrary number is only used one time in a cryptographic communication. [12]
- **Decryption:** Recipients decrypt the masked header using their node ID and verify the protocol ID to process the packet correctly.

Remote Procedure Calls

Discv5 enhances node discovery and communication through several RPCs. It removes FIND_VALUE & STORE commands introduced by Kademlia [19], leaving only necessary RPCs for node lookups and adding other Calls needed for communication between nodes [26]:

- **PING & PONG:** Nodes periodically send PING messages to check the liveliness of other nodes, sharing their ENR sequence number. The PONG response confirms availability and shares the recipient's current status and contact details.
- **FINDNODE:** Requests information about nodes at specific distances in the network to maintain connectivity.
- **NODES:** Responds to FINDNODE with a list of nodes that match the search criteria, aiding network discovery.
- **TALKREQ & TALKRESP:** Facilitate custom data exchange between nodes for specific applications, with TALKRESP providing the required response or indicating unsupported protocols.

There are other methods that have not been cited for the purpose of this thesis as it is a practical exploration, while these methods are yet to be implemented and only cited by Ethereum in theory. They are a work under development, but will enhance the protocol by providing a sub-protocol advertisement system through several RPCs such as REGTOPIC, TOPIC QUERY... [26] In essence, these RPCs would be interesting to implement in order to have a distributed network that is future-proof, allowing several sub-protocols to be under the same network, by making an advertisement mechanism to further propagate nodes depending on which sub-protocols they maintain for networking.

2.3.5 Potential Goals of Discv5

General Goals

Discv5 introduces several solutions and key concepts that can be found under their official rationale as well [24]:

- Like stated earlier, the verification process proposed by discv4 was unreliable. The FIND_NODE request may fail when node A assumes that node B already knows about a recent PING/PONG interaction.

- In discv4, a node can provoke a response from any other node simply by knowing its ip address. This can be abused by malicious nodes and create DDOS (Distributed Denial Of Service,when multiple services flood a single targeted system with requests, forcing it to shutdown) attacks. Making it a requirement to know the node's ID and making it expensive to obtaining it helps mitigating this sort of attack.

- Allowing more node ID cryptosystems, which in turn will result in a more interoperable node discovery system that is future-proof and supports the idea of a multichain for Ethereum for instance. For now (discv4), only secp256k1/keccak256 is supported.

- Replacing node information tuples with ENRs. As discussed earlier, ENRs can provide many arbitrary information about a node, which can permit several new abilities such as permitting capability advertisement to other nodes and transport negotiation, meaning that transport protocols can differ from node to another and be classified accordingly with the help of topic advertisement.

- Guard against Kademlia implementation flaws: Since Kademlia operates based on distance metrics between nodes calculated following a distance metric, a misuse of this metric can result in a fragmentation of the network.

- Support for finding nodes based on an arbitrary topic identifier. This, according to the rationale, should be as fast or even faster than the traditional approach to node querying.

- While Discv4 relies on nodes joining the network with a clock on the machine that is very accurate to prevent replay attacks, where a node would receive requests and try to simulate false packets in the network, It is being prevented poorly in discv4, because it led to many complaints when joining the network or with connectivity problems. The Ethereum foundation took this chance of upgrading the discovery protocol to also introduce a solution to this common issue in Discv5 [20].

- The traffic between nodes should be obfuscated, meaning making it hard to identify/recognize patterns or read data.
  This is done to avoid sniffing (the process of "hearing" packets that are being sent through the network).

Security Goals

The security goals of discv5 are intended to add a layer of security to how Kademlia operates to avoid the node discovery protocol from being compromised by slowing it, polluting the routing table of a node with false information, or directing a lot of traffic to a victim node.
These goals can vary and come as follows, as described per the Discv5 Rationale [24] :

- Replay attacks [15]:
  - Replay of the handshake: A handshake (process of nodes that first get in touch with each other) if successfully replayed, may be utilized to propagate false information to the node receiver that would then store old information to its routing table
  - Replay NODES:
    This would pollute the routing table of the receiving node with stale information as well
  - Replay PONG: A replayed pong can convince a node receiver that the sender is alive when it isn't, leaving it in the routing table instead of eliminating it by replacing it with an operating node.
- Kademlia redirection: Since the node lookup algorithm in kademlia can alter the routing table of each node in the process, containing false information intended at directing traffic, such as IP-addresses, can alter routing tables and pollute them, which would result in fake endpoint information and can cause excess traffic to a victim node, isolating it from the network. This process can also be further propagated by the node receiver, causing further disturbance in more nodes.
- Polluting a node's routing table can also be made through unsolicited replies by mimicking past legitimate replies in weak implementations.
- Amplification: Small messages can be sent to malicious IP addresses that have been altered through polluting a routing table. These receiving malicious nodes can amplify the messages, directing larger responses to digest to victims
- Direct validation of a newly discovered node can be an attack vector. A malicious node can supply wrong node information with the IP of a victim node. The validation traffic is then directed at the victim.
- Kademlia ID count per address validations: A malicious node is able to supply many logical node Ids to a single IP address, which in turn will cause the correct node holding that IP address from joining the network because most of the logical node IDs tied to that specific IP address have been issued.

- Sybil / Eclipse attacks: By having control over many real nodes or by spoofing many logical node IDs for a small number of physical endpoints, an attacker may be able to isolate an area of the network that can prevent newcomers to join the actual valid network, or to manipulate their traffic, by directing them to that part of the network [15].

2.3.6 Features and Mechanics of Discv5

Having outlined the strategic goals and security enhancements of Discv5, we now transition to a detailed examination of its operational features and underlying mechanics. This section will delve into the theoretical constructs and practical applications that define Discv5, highlighting how these contribute to a more secure and efficient node discovery protocol within the Ethereum network. First, here is an overview of the changes in comparison to discv4 before diving deep into the theory behind it:
In comparison with discv4, we can have the following list of changes according to Ethereum's official Github repository [23]:

- **Introduction of Topic-based Advertisement**: Nodes can now advertise their services based on specific topics, enhancing the relevance and efficiency of node discovery.
- **Arbitrary Node Metadata Storage**: Nodes can relay and store metadata, allowing for richer and more dynamic node information, allowing nodes to connect to each other based on common sub-protocols.
- **Flexible Node Identity**: The protocol supports multiple cryptographic keys, reducing dependence on secp256k1 keys.
- **Independence from System Clock**: Communication and operational integrity no longer rely on the accuracy of the system clock upon joining the network.
- **Encrypted Communication**: All node communications are encrypted, protecting against passive observation.

Bootstrapping Process & Handshake Mechanism

One of the key aspects of secure communication between nodes Discv5 is how messages are encrypted and structured. The Discv5 protocol does not send messages as plain text; instead, it uses a shared session key to encrypt them. This session key is established through a handshake mechanism.
When a node first starts and joins the network, it must be pre-loaded with a list of ENRs. This is the bootstrapping process. And these nodes are also called bootstrap nodes. They are chosen based on their reliance among the system. For instance, The Ethereum Network provides a set of bootnodes to new peers that they can use to bootstrap their node [5].

Any node can be chosen to be a bootstrap node, active or not.
To begin the process, the joining node sends a PING request to these bootstrap nodes to check their availability and status. To facilitate secure communication within the Discv5 protocol, nodes first need to establish a common set of cryptographic session keys. This is accomplished through a handshake process, which utilizes an authenticated key agreement protocol that employs elliptic curve cryptography. This series of steps allows nodes to verify each other's identities and generate the essential cryptographic keys required for protected communications. The handshake is a critical component of Discv5, safeguarding against unauthorized access to exchanged messages, securing sensitive data, and enhancing the overall reliability of node interactions.

Steps in the Handshake Mechanism [25]

Let's suppose we have nodes A and B and that A wants to communicate with B, as described in 8. Since the communication between two nodes is encrypted with asymmetric keys, node A will check if it has the session keys stored from past interactions. If it doesn't, it will initiate the handshake by sending random content to node B 8 (first step) to signal that a new secure session is needed.
In response, Node B sends a challenge in the form of a WHOAREYOU packet 8. (Step 2) This includes a unique 64-bit nonce generated randomly and the sequence number of node A's record if available. The nonce value helps Node A to trace the challenge back to the initial request packet, while the sequence number allows Node A to determine if it needs to send an updated record to Node B. it proceeds by sending back the FINDNODES request to Node B, including an ID signature and a temporary public key for secure key agreement. The nodes use the public and private keys shared during the handshake to perform a Diffie-Hellman key agreement. They then use HKDF to derive session keys for secure communication. They proceed by completing the handshake in a last step [25]:

- Node B receives the handshake message, verifies the ID signature, and uses the session keys to decrypt the message. If successful, it sends a NODES response encrypted with the recipient's session key.
- Node A decrypts and authenticates the NODES response using its session keys, completing the handshake.

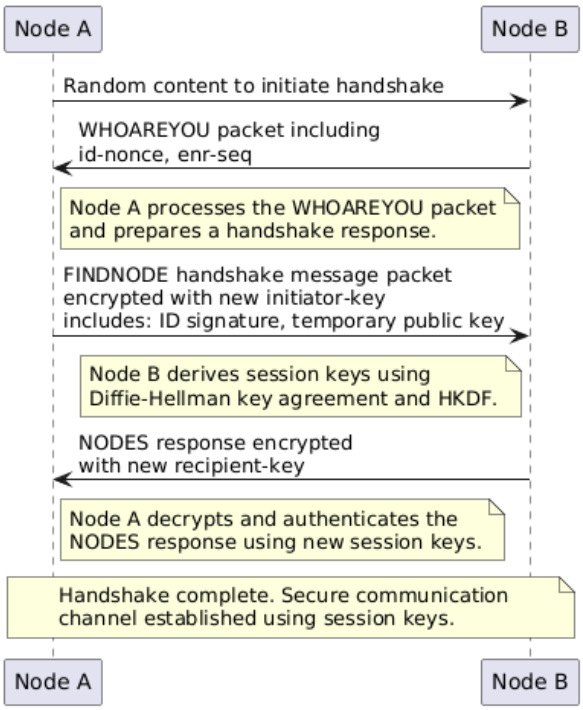


Figure 8: Example Handshake Mechanism between Node A & B in Discv5

Kademlia-inspired Routing Table & Record Management

As described across the thesis, disc is in essence heavily inspired from Kademlia-DHTs. It does so by neglecting the storage aspect of it and by putting a focus on how the lookup algorithm works.
In Discv5, just like in Kademlia, the routing table of a node is structured as buckets as described in the section 2.2.4. Each bucket can contain up to 16 node records (ENRs) . [25] Each node can have up to 256 buckets in its routing table, with each bucket corresponding to a different distance range from the node, measured by the XOR distance of their public keys. These buckets are ordered by increasing distance from the node, making the first bucket contain the closest neighbouring nodes, and the last bucket containing the furthest ones. Within each bucket, nodes are sorted based on last time seen as described in 2.2.6, allowing the nodes to keep track of which neighbouring nodes are the most active, for efficient routing. Finally, the buckets are maintained up to date as nodes leave and join the network and by periodically performing node lookups to check their availability and update the buckets in accordance to their responses.

Conclusion

As we conclude the theoretical exploration of the Discv5 discovery protocol, we have observed its sophisticated mechanisms designed to facilitate efficient and equitable discovery processes within decentralized networks. The protocol's advertisement system

is however still not implemented practically and remains a theory for now.
The transition from this theoretical framework to the practical implementation phase of a decentralized application utilizing Discv5 represents a pivotal shift. Moving forward, the insights gained from the protocol's design and its mechanisms for managing node interactions will be instrumental. These insights will guide the development of scalable decentralized applications, ensuring they leverage Discv5's strengths.
This next phase will involve applying the theoretical knowledge to real-world scenarios, focusing on the integration of Discv5's discovery functionalities. By bridging theory with practice, the forthcoming implementation aims to explore the protocol's efficacy and integration.

3 Discv5 in Practise

3.1 Introduction

With a solid theoretical foundation laid in the preceding chapters, this segment of the thesis marks a pivotal transition into the practical application of the Discv5 protocol within a decentralized application environment.
The practical application of the theoretical concepts discussed earlier will be showcased through a detailed system design and implementation section. This will include:

- **Architecture:** A comprehensive blueprint of the system architecture, detailing the integration of Discv5 within a decentralized setting akin to an image board platform
- **Functionalities:** An exploration of the core functionalities implemented, demonstrating how Discv5 enhances application responsiveness and network reliability.

Finally, we will evaluate the system implemented through rigorous testing, providing real-time examples of how discv5 is being used within the implementation, while assessing the usability, effectiveness and potential areas for improvement.

3.2 Exploring the Rust Discv5 Library

3.2.1 Overview

The discv5 library, or in Rust terms "crate", is essential to explain in order to build an application with it. The protocol is split into 4 main layers [7]:

- **Socket:** It is responsible for the underlying UDP socket of the protocol. It generates individual tasks for sending and receiving packets with encoding and decoding, from the UDP socket.

- **Handler:** Takes care of the session establishment and encryption between nodes upon first contact as described in the handshake procedure in figure 36. The protocol's communication is encrypted with AES_GCM. The creation and maintenance of sessions between nodes and the encryption/decryption of packets from the socket is realised by the handler::Handler struct running in its own task.

- **Service:** Contains the protocol-level logic. The service::Service manages the routing table of known ENR's, and performs parallel queries for peer discovery. It also runs in it's own task.

- **Discv5:** The application level. Manages the user-facing API. It starts/stops the underlying tasks, allows initiating queries and obtaining metrics about the underlying server.

Conveniently, the crate provides access to the stream of Events occurring on the protocol. This stream can be obtained from Discv5:event_stream.
The discv5 crate is composed of 8 different modules according to the documentation, which can be divided into 2 different main categories: The handling of the Networking, and the handling logic of disc. The logic handling has the following modules:

- **Handler**: As described earlier, this contains the session and packet handling.

- **kbucket**: This module is internalized and not available at the surface level. It is the implementation of the Kademlia routing table as used by a single peer within a Kademlia DHT.

- **rpc**: The Remote Procedure Calls occurring between nodes are handled in this module. 3 different structs are present in it, namely the Request (sent between nodes), the RequestId (A type to manage the request IDs), and a Response, a struct sent as a response to a Request.

- **service**: The discovery protocol V5. This is the core of the crate, where our main Events are happening. Find_node, TalkRequest, Handshake procedures, frequent node updates, ENR storage and querying is happening in this module. According to the documentation, ENRs are not automatically stored in the routing table, and only established sessions are added to it in order to avoid storing inactive node information and having only valid ENRs added. However, manual additions can be added with a function add_enr(). Response to queries involving finding nodes return PeerIds, and only those of which an established session has been made with. They have their ENRs returned to store them in the sender's routing table, and are returned only when they have a valid address to connect to (IP Address and Port). Other untrusted Peer Ids with no session established can be discovered as well by firing another event from the Service:Discovered event, which is basically fired when peers get discovered.

3.2.2 Discv5 API Overview

Following the overview of the library, we will now focus solely on the Discv5 application level documentation that is stated in 3.2.2. It can be found under [9]. We can dissect the main functionalities that are exposed to the application level as displayed in figure 9.
The discv5::Discv5 is the main service struct, providing the user-level API for performing queries and interacting with the underlying service.
It provides the main discv5 functionalities that we have covered in the Section 2.3.4.
It takes the following parameters:

- **enr**: An ENR generated from CombinedKey. There are two different signing keys that can be used. See figure 9.
- **enr_key**: The signing key that has been used to generate our ENR.
- **Config**: a Pre-set of constants that is of type Config. There is a default configuration that can be used as well, see 9. The Config is generated using ConfigBuilder, which allows us to read and write to our configuration.

Methods

The Discv5 struct methods cover all the primary RPC methods that we have cited in section 2.3.4. We can categorize these methods covered in our figure 9 as such:

- Administrative functionalities
  - **Start function**: Starts the required tasks and begins listening on a given UDP SocketAddr.
  - **Shutdown**: Terminates the service.
- Node Discovery & Management
  - **add_enr**: Adds an ENR to the routing table.
  - **remove_node**: Removes a node ID from the routing table.
  - **disconnect_node**: Marks a node in the routing table as Disconnected.
  - **ban_node**: Bans a node from the server for a specified duration or permanently.
  - **ban_node_remove**: Removes a banned node from the banned list.
  - **permit_node**: Permits a node, allowing it to bypass the packet filter.
  - **permit_node_remove**: Removes a node from the permit list.
- Informative functionalities
  - **nodes_by_distance**: Returns a vector of nodes closest to specified distances.

  - **connected_peers**: Returns the number of connected peers in the routing table.
  - **metrics**: Retrieves metrics associated with the server.
  - **raw_metrics**: Exposes the raw reference to the underlying internal metrics.
  - **local_enr**: Returns the local ENR of the node.
  - **external_enr**: Returns the external ENR, which is the local ENR exposed via an Arc.

- ENR Manipulation
  - **enr_insert**: Inserts or updates a field in the local ENR.
  - **update_local_enr_socket**: Updates the local ENR TCP/UDP socket settings.

- Routing Table Manipulation
  - **kbuckets**: Returns the routing table of the Discv5 service.
  - **table_entries_id**: Returns an iterator over all node IDs in the routing table.
  - **table_entries_enr**: Returns an iterator over all ENRs of nodes in the routing table.
  - **table_entries**: Returns an iterator over all entries in the routing table.
  - **with_kbuckets**: Uses a closure to interact with the kbuckets for potential viewing or modifying.

- Network Security Policies
  - **ban_ip**: Bans an IP address from the server for a specified duration or permanently.
  - **ban_ip_remove**: Removes a banned IP from the banned list.
  - **permit_ip**: Permits an IP, allowing all packets from it to bypass the packet filter.
  - **permit_ip_remove**: Removes an IP from the permit list.

- Discv5 Remote Call Procedures (RPC)
  - **send_ping**: Sends a PING request to a node.
  - **talk_req**: Sends a TALK request to a node using specified protocol and request data.
  - **find_node_designated_peer**: Sends a FINDNODE request to a designated peer based on specified distances.

  - **find_node**: Initiates an iterative FIND_NODE request targeting a specific node ID.
  - **find_node_predicate**: Starts a FIND_NODE request based on a given predicate.

- Event Capturing
  - **event_stream**: Creates an event stream channel for receiving Discv5 events.

The CombinedKey in 9 file provides currently only ‘secp256k1‘ and ‘ed25519‘ key types support, as per the API coverage in section 3.2.2. The file provides a keypair generation functionality for both key types, as well as a function to be able to encoding and decoding them from/into bytes. Finally, it also provides a function to derive the public key from it, as well as an essential function named **verify_v4()** which takes as parameters a byte encoded message and a signature of the message, in order to verify that the message has been signed by the keypair itself.
As we wrap up our detailed exploration of the Discv5 library and its API coverage, we've established a solid foundation for practical application. The insights gained provide us with the ncessarz knowledge to embark on developing an application with it.

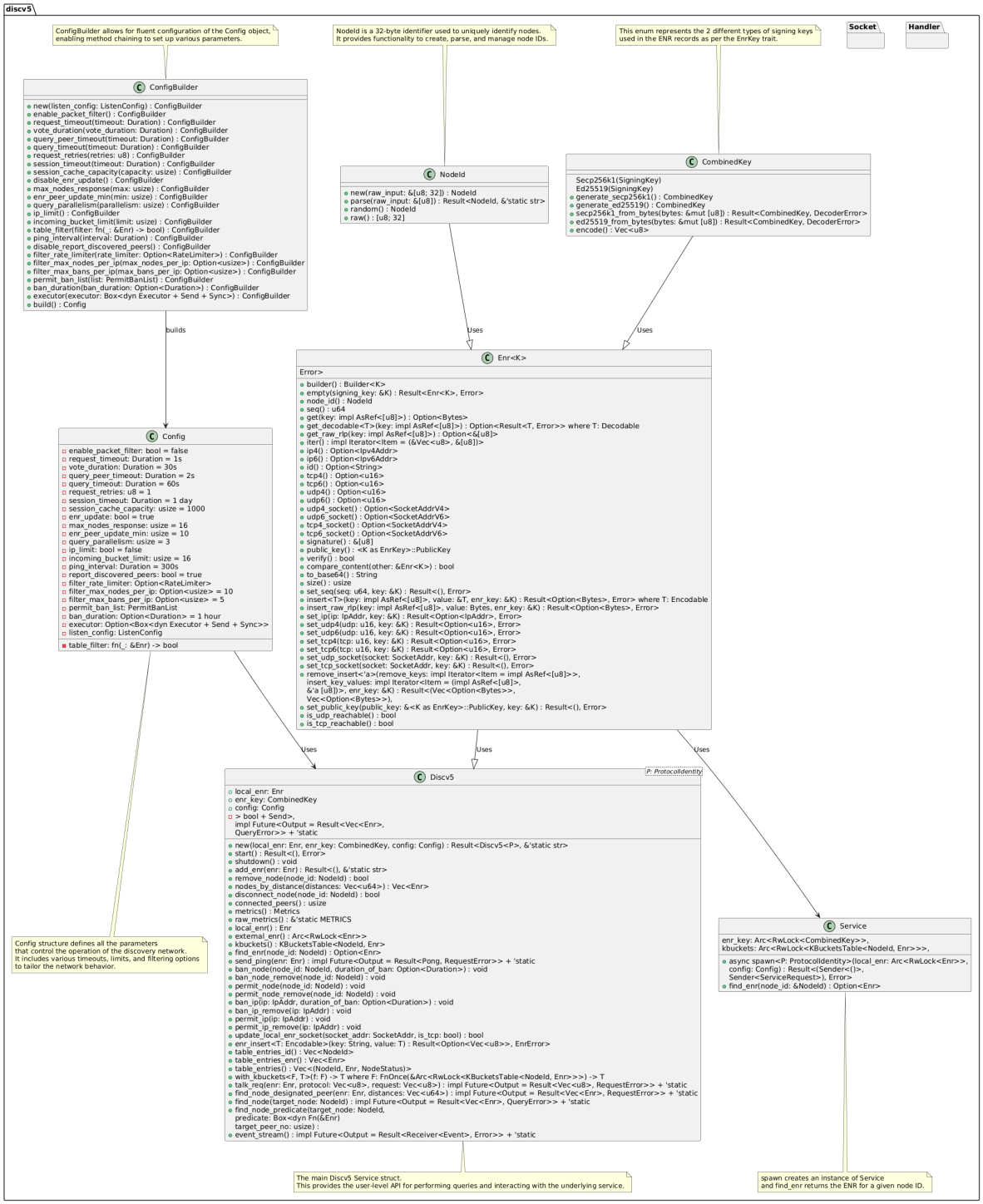


Figure 9: Discv5 API Overview

3.3 Analysis

Now that we have covered the API of Discv5, we set a clear stage on how we can make use of it to develop our application.
We will start by uncovering problems that can come up when making a decentralized application, and that need to be generally conscious of. We will then present the necessities for our application in the application environment section.

3.3.1 Problem Analysis

Distributed Applications present unique challenges that impact their design, efficiency, and security. Some of the primary challenges include:

- Centralized risks: Despite the decentralized intent of p2p systems, improper implementation can inadvertently reintroduce centralization, diminishing the system's resilience and control distribution.
- Scalability vs speed: While distributed systems inherently offer excellent scalability, this can sometimes come at the cost of reduced performance, especially in terms of transaction or operation speed.
- Security vulnerabilities The open nature of distributed systems can expose them to various security threats, making comprehensive security measures essential.

These challenges are intrinsic with the core functionalities of Discv5 and the Kademlia Distributed Hash Table (DHT), as these technologies aim to optimize the balance between decentralization, scalability, and security.

3.3.2 Application Environment

This thesis aims to showcase how discv5 can be implemented within the scope of an application. For those reasons, its focus will be more on the technicality of how discv5 can be utilized, rather than how the system actually works.
To put our focus more on that point, we will be making a simple application as described in the motivation section, which is a simple data storage system that can serve for social networking purposes, in a decentralized manner.
The applications consists of Posts and Comments. Under these topics, people are able to make posts where they write what they wish to. Every post can only be present under one single topic at a time. Posts can have comments, which present the same structure as the posts themselves. The difference being, that a post is mapped to a topic, whereas a comment is mapped to a post / comment. In a coding perspective, posts and comments are the same. Except that their keys vary. This can be well represented in a key-value storage system.
For the next chapter, we will discuss various solutions on how to distribute this data across the network, as well as how peers can find each other and maintain an overlay

network with discv5. Some architectures that can come to mind, based on the technologies that we have covered, will be presented. We will then decide on which one is better with a comparative analysis following a process of elimination, and proceed with the implementation of the better solution.

3.4 Solution Propositions

Upon uncovering the necessities for our application, we will propose 3 solutions that leverage the use of Discv5 for the networking layer between nodes. We will then move along with a process of elimination to decide which system fits best and move along with its implementation, demonstrating how Discv5 can be utilized along the process.

Topic-based Node Discovery & Clustering

This solution organizes a network by clustering nodes around specific topics of interest to enhance content discovery efficiency and system responsiveness. Nodes indicate their interest in topics when they join or through dynamic updates, participating in a voting mechanism that determines new topic additions. Nodes are grouped into clusters based on shared interests, and queries or content are directed only towards relevant clusters, not the entire network.
Topics are established through a governance consensus, allowing nodes to vote on potential new topics. Each node's Ethereum Node Record (ENR) contains key-value pairs representing topics they've voted on, aiding in topic-based node discovery and efficient query responses. The TOPICQUERY RPC is used to connect nodes under the same topic, and nodes broadcast changes in their topic interests, facilitating real-time updates and maintaining a responsive and interconnected network.

Federated Server Model with discv5

In this model, the application is powered by a set of federated servers, each serving different sectors of the application. In this system, any node can join with a chosen topic and maintain a governance over the topic it is hosting. For those reasons, they are called federated servers; hosting different parts of the application. Each server in turn maintains its own centralized database. All servers are inter-connected for peer discovery using the Discv5 routing table.

Decentralized P2P Messaging System

This system uses a peer-to-peer (P2P) architecture where each user operates as a node within the network. The nodes communicate directly with each other without a central server, using discv5 for node discovery and maintaining a dynamic, decentralized network.
This distributed network maintains different layers that are partitioned across the entire system:

- Networking layer: The system maintains a networking layer using discv5 that is solely reserved for node discovery. It makes node lookups periodically to maintain fresh up to date routing tables with reachable nodes.

- Storage layer: This layer will be in charge of handling the data. It utilizes an implementation of the Kademlia DHT. Discv5 will pass the discovered nodes to store them under the DHT, and external endpoints are exposed to the client in order to interact with the distributed network by storing and retrieving key-value pairs to the DHT. Content such as images and posts can be addressed by hashes, which ensures data integrity and aids in retrieving data from the DHT. To ensure fault tolerance, the routing table of the discv5 server is refreshed often, handing node information to the DHT.

3.5 Evaluation

This section will detail each proposed solution, explaining how they address specific challenges of our decentralized application to deduct which solution fits best the incentive of this thesis.

3.5.1 Federated Server Model with discv5

Nodes are regrouped by the discv5 protocol, connecting them between each other and data can be fetched and passed from one node to another using the TALK request from the discv5 library, however, every node would represent a form of authority over a specific topic, which is not decentralization per definition. It presents high risks against fault tolerance: If a certain node is shutdown, a whole topic is shutdown and its data is forever lost. This system is only decentralized when it comes to networking, but the application layer itself is centralized.

3.5.2 Topic-based Node Discovery & Clustering

This method organizes nodes into clusters based on the topics they are interested in. However, there are practical limitations in the current implementation of the discv5 protocol that affect this method. Key functionalities such as the REGTOPIC RPC and the TOPIC management system are theoretically described but not currently recommended for use, as highlighted in the official Discv5 documentation (limitations referred in section 2.3.4 of our thesis). Additionally, this system extends the use of the Ethereum Name Record (ENR) beyond its typical networking functions, which could introduce complications. Since discv5 is designed primarily for networking, using it extensively for other purposes may lead to overhead issues and complicate the maintenance of routing tables. Ideally, a distributed network should treat all nodes uniformly, abstracting the application layer from the client.

3.6 Decision & Justification

After careful consideration of the proposed solutions for implementing a decentralized application with discv5, and a careful review of how each solution can operate, it goes without saying that the latest solution, the Decentralized P2P Messaging system, suits best the narrative of a structured decentralized system for our application.
This system optimally leverages the inherent features of discv5 and Distributed Hash Tables (DHT) to address fundamental requirements of decentralization. The system's architecture is inherently decentralized with no single point of failure, as all nodes find each other and operate equally with no central authority. The use of Kademlia DHT enhances the system's scalability. As the network grows, the DHT efficiently manages the increasing number of nodes and data entries through its proven distributed network algorithms. The introduction of discv5 as the networking layer to the system will ensure that we have secure sessions established between nodes, while having a fast-paced routing mechanism for nodes to discover each other and communicate between one another securely.
This model not only meets the technical requirements of the system but also aligns closely with the ideological goals of maintaining network decentralization. This justification concludes the decision to adopt the Decentralized P2P Messaging system as the preferred architecture for the application, providing a solid foundation for future development.

4 Example Application

This chapter describes the system design of our application presented in the latest solution, the goal being to showcase how we can utilize Discv5 and an implementation of its API. It has been decided that it will incorporate Discv5 for node discovery and for networking purposes only. Furthermore, it will make use of a DHT system that is in contact with disc to store the nodes across the network within it, as well as data storage management and retrieval across the nodes.
We begin with an exploration of the foundational functionalities of Discv5, demonstrating how basic operations are seamlessly integrated to provide initial network discovery and connectivity. The discussion will then expand to encompass advanced features of Discv5, evaluating how it can be integrated with our Kademlia DHT.

4.1 Architecture

The architecture of the decentralized data storage leverages a multi-layered approach. Each layer featuring specific functionalities to the application.

As we can see in figure 10, These layers can be grouped into the following, following the Osi-Model 2.1.2:

- **Networking Layer:** Utilizes the discv5 protocol for peer discovery and network

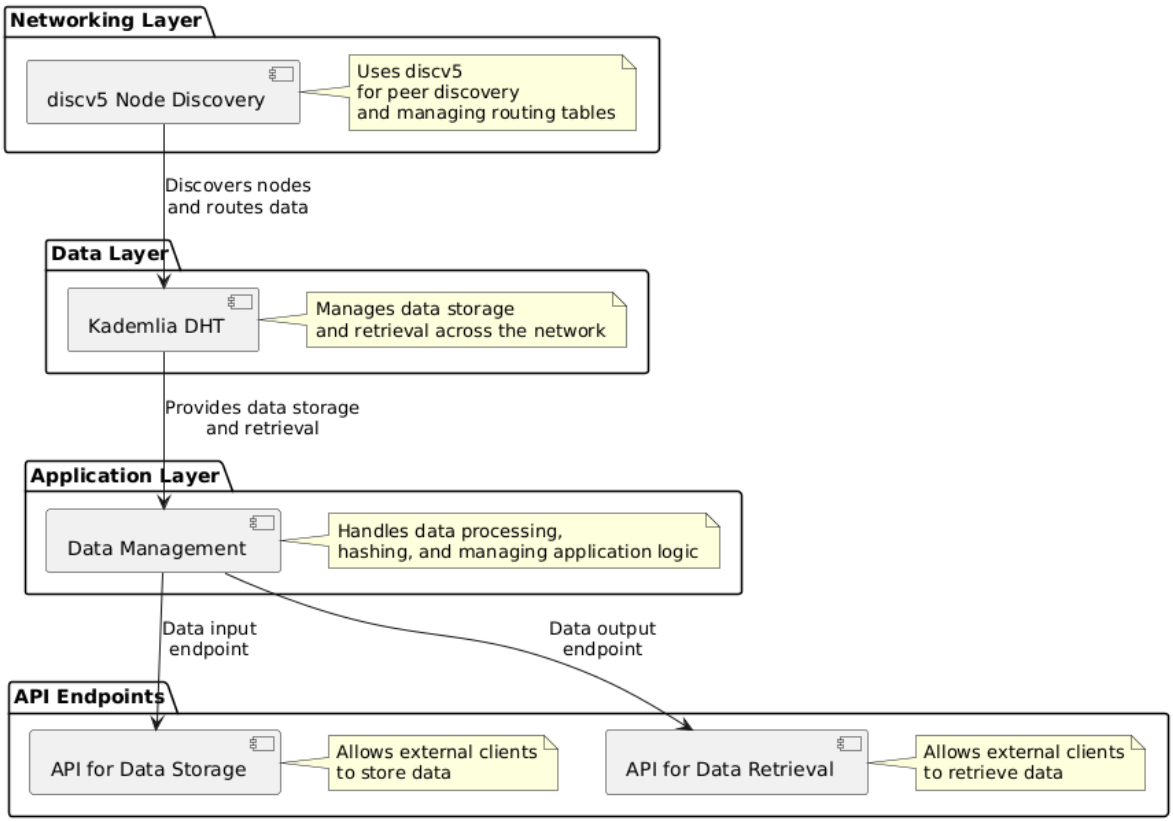


Figure 10: Overview of the Application Layers

topology maintenance, ensuring secure node connections via a handshake protocol and propagating node information through Disc.

- **Application Layer:** Employs a Kademlia Distributed Hash Table (DHT) for decentralized data storage and retrieval. This layer optimizes data routing by maintaining current node information from the Networking Layer, leveraging node ENRs for efficient data distribution across nodes.
  Handles data processing and management, incorporating Data Management components for hashing and data preparation prior to DHT storage. It offers API Services that act as interfaces for client interactions, supporting operations like content posting and retrieval, thus abstracting system distribution complexities from end-users.

Inter-layer communication within the architecture (see Figure 10) enhances system cohesion and functionality:

- **From Networking to Application Layer:** Networking Layer supplies node updates to the Data Layer, facilitating DHT routing optimizations based on the live network state provided by the Discv5 protocol.

- **APIs and Data Management:** Application Layer's APIs interface with external clients for data transactions, with the DHT.

The separation into distinct layers allows for focused scalability and maintenance, with each layer designed to independently handle its specific set of responsibilities within the broader system context.

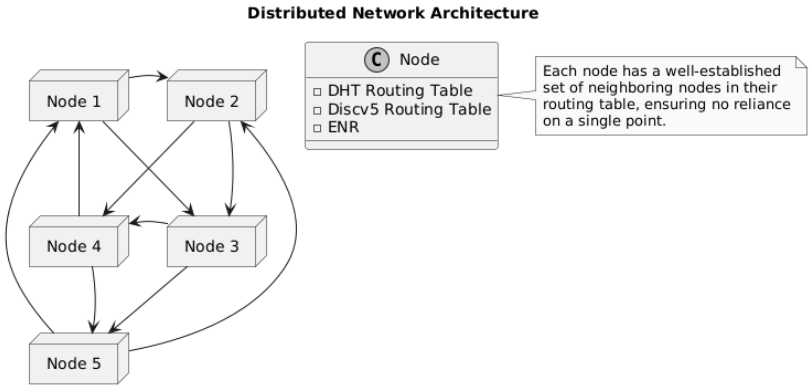


Figure 11: Distributed Network Architecture

This architecture as presented in figure 11, supports a decentralized model by eliminating central points of failure and distributing data and functionality across a wide network of nodes. It is designed as a **distributed network** per definition in section 2.1.1, as no complete reliance to a single point is required. Furthermore, we can classify this networking system as **structured pure p2p** as described in Section 2.1.3. No single node has any extra authority and all nodes are equal in regards to networking, having each a well established set of neighbouring nodes in their routing tables that are optimized with Discv5.

4.2 Discv5 Functionalities Code Implementation

Upon covering the general architecture of our application, let's have a deep dive into how it works by going through the functionalities of it, how the code works, describing its workflow and how it operates. We will begin by covering how the bootstrapping process described in figure 12 can be implemented practically, to then rotating over the RPC methods from discv5, as cited in section 2.3.4 one by one, with a thorough code explanation.

4.2.1 Starting the Discv5 server

The server runs with the following command:

```
cargo run -- --enr-ip4  <ip-address> --port <port>
```

It takes the IP address and Port as arguments, which will be essential to generate an ENR containing these information. An enr_key is generated using the function from discv5 CombinedKey::generate_secp256k1(), which creates a new cryptographic key for the node using the secp256k1 elliptic curve algorithm. This key is used to sign and verify the node's identity and to then establish communications securely, as discussed in 2.3.2. This new fresh enr key is then immediately used to build the Ethereum Node Record of the Node. The enr variable is then assigned by calling build_enr() with the previously derived arguments, the generated key, and the two distinct ports (port and port6). The function build_enr constructs an Ethereum Node Record. This record includes information such as the node's IP address, ports, and its public key, making it discoverable and verifiable by other nodes in the network. We finally make use of both the ENR key and the freshly generated ENR to build our discv5 server. Here is an overview of the code to reach this process :

```
let args = parse_args();
let port = args
    .port
    .unwrap_or_else(|| (rand::random::<u16>() % 1000) + 9000);
let port6 = args.port.unwrap_or_else(|| loop {
    let port6 = (rand::random::<u16>() % 1000) + 9000;
    if port6 != port {
        return port6;
    }
});

//Generating fresh ENR for node
let enr_key = CombinedKey::generate_secp256k1();

let enr = build_enr(&args, &enr_key, port, port6);
// the address to listen on.
let listen_config = match args.socket_kind {
    SocketKind::Ip4 =>
    ListenConfig::from_ip(IpAddr::V4(Ipv4Addr::UNSPECIFIED), port),
    SocketKind::Ip6 =>
    ListenConfig::from_ip(IpAddr::V6(Ipv6Addr::UNSPECIFIED), port6),
    SocketKind::Ds =>
    ListenConfig::default()
        .with_ipv4(Ipv4Addr::UNSPECIFIED, port)
        .with_ipv6(Ipv6Addr::UNSPECIFIED, port6),
};

// default configuration with packet filtering
let config = ConfigBuilder::new(listen_config)
```

```
        .enable_packet_filter()
        .build();

    info!("Node Id: {}", enr.node_id());
    if args.enr_ip6.is_some() || args.enr_ip4.is_some() {
        // if the ENR is useful print it
        info!(
            base64_enr = &enr.to_base64(),
            ipv6_socket = ?enr.udp6_socket(),
            ipv4_socket = ?enr.udp4_socket(),
            "Local ENR",
        );
    }

    //Initializing discv5 server
    let mut discv5 = start_discv5_service(enr, enr_key, config).await;
```

As we can see from the following code , the build_enr function takes the necessary information to be stored under the ENR such as the IP address and port, as well as the newly generated key, and makes use of the enr::Enr::builder() struct provided by the discv5 library. This is also the process where we can add some arbitrary data to our ENR as seen in section 2.3.2. Since we do not need any information for our simple implementation, we are simply passing the IP address and the port. We finish this process by executing the builder.build() function, to which we are passing the previously generated keypair.
The ENR builder functionality is :

```
pub fn build_enr(
    args: &FindNodesArgs,
    enr_key: &CombinedKey,
    port: u16,
    port6: u16,
) -> enr::Enr<CombinedKey> {
    // Clone the key to use in the ENR builder
    let mut builder = enr::Enr::builder();
    if let Some(ip4) = args.enr_ip4 {
        if ip4.is_unspecified() {
            builder.ip4(Ipv4Addr::LOCALHOST).udp4(port);
        } else {
            builder.ip4(ip4).udp4(port);
        }
    }
    if let Some(ip6) = args.enr_ip6 {
```

```
        if ip6.is_unspecified() {
            builder.ip6(Ipv6Addr::LOCALHOST).udp6(port6);
        } else {
            builder.ip6(ip6).udp6(port6);
        }
    }
    // Pass the cloned CombinedKey directly to the build method
    builder.build(enr_key).unwrap()
}
```

Our discv5 server is now successfully setup and ready to operate. We need to be able to connect it with the other nodes within our distributed system.

4.2.2 Bootstrapping the Discv5 Server

The server starts by defining a hard-coded set of nodes that can serve as bootstrap nodes. We do this by providing their ENRs in a bootstrap.json file. If there is no ENR stored in this file, the node will start without connecting to any other node. It can therefore serve as a bootstrap node to initiate the distributed system as a whole.

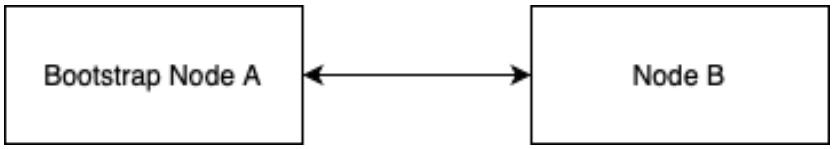


Figure 12: Bootstrap Node

We can think of the bootstrap process as the most sensitive part about the system, because it is the only part of the implementation where we need to rely on trusting the node we are assigning as the bootstrap node, as discussed in section 2.3.6.
Going back to our code, we can find the bootstrap process as follows :

```
#[derive(Debug, Clone, Serialize,
Deserialize, PartialEq, Eq, Ord, PartialOrd, Hash)]
pub struct BootstrapStore {
    /// The list of bootstrap nodes.
    pub data: Vec<BootstrapNode>,
}

#[derive(Debug, Clone, Serialize,
Deserialize, PartialEq, Eq, Ord, PartialOrd, Hash)]
pub struct BootstrapNode {
    pub enr: String,
}
```

```rust
pub async fn boostrap(discv5: &mut Discv5, file: Option<String>)
-> eyre::Result<()> {
    if let Some(f) = file {
        // Read the JSON bootstrap file
        println!("Current directory: {:?}", env::current_dir()?);
        info!("File : {}", f);
        let file = File::open(f)?;
        let reader = BufReader::new(file);
        let bootstrap_store: BootstrapStore = serde_json::from_reader(reader)?;

        // For each bootstrap node, try to connect to it.
        for node in bootstrap_store.data {
            // Skip over invalid enrs
            if let Ok(enr) = Enr::from_str(&node.enr) {
                let node_id = enr.node_id();
                match discv5.add_enr(enr) {
                    Err(_) => {
                    log::warn!("Failed to bootstrap node with id: {node_id}")
                    }
                    Ok(_) => {
                        info!("Bootstrapped node: {node_id}");
                    }
                }
            }
        }
    }

    Ok(())
}
```

We are deriving the ENRs from the bootstrap.json file. This function takes the file name as a parameter, as well as the discv5 Server. It begins by reading and parsing the JSON file into a list of ENRs that can serve as bootstrap nodes using serde. Serde, a serialization and deserialization framework, is critical for efficiently handling data exchange between nodes in a distributed system, allowing complex data structures to be easily converted to and from a communicable format. [8] It then proceeds to extract the node Ids from the ENRs using the enr.node_id() function provided by the library, iterating over each ENR and adding it to the discv5 known ENRs using discv5.add_enr() function to add them to the known ENRs list of our routing table, successfully bootstrapping the discv5 server and adding known servers to its local routing table.

```rust
    let bootstrap_file = Some("bootstrap.json".to_string());
    // if we know of another peer's ENR, add it known peers
```

```
// -> Bootstrap process
boostrap(&mut discv5, bootstrap_file.clone()).await.unwrap();

// start the discv5 service
discv5.start().await.unwrap();}
```

Right after bootstrapping the Discv5 server, we finally start it with the discv5.start() function. These ENRs will then go through the handshake process, that will lead to the establishment of a secure connection between both nodes for further data exchange between them, as described in figure 36.
To put this into action, we have established a secure connection between two nodes using a local computer and a Virtual Private Server.
After kickstarting the code on Node A, we received the ENR printed on the terminal. We have taken this ENR and pasted it within the list of bootnodes in bootrap.json for the Node B, locally. Both nodes have successfully found each other and established a secure connection as depicted in the following pictures, where we can see the terminal printed results.

Figure 13: Node A: Bootstrap Node

4.2.3 Node Discovery

Now that our nodes are able to find each other, we are performing a constant node lookup over the network periodically, separated by 30 seconds, to always have up to date nodes.
This is executed by the following code, alongside with Tokio to spawn it on a separate thread in order to not block the main function from running. Tokio, an asynchronous runtime for Rust, provides the tools necessary to build scalable and non-blocking I/O

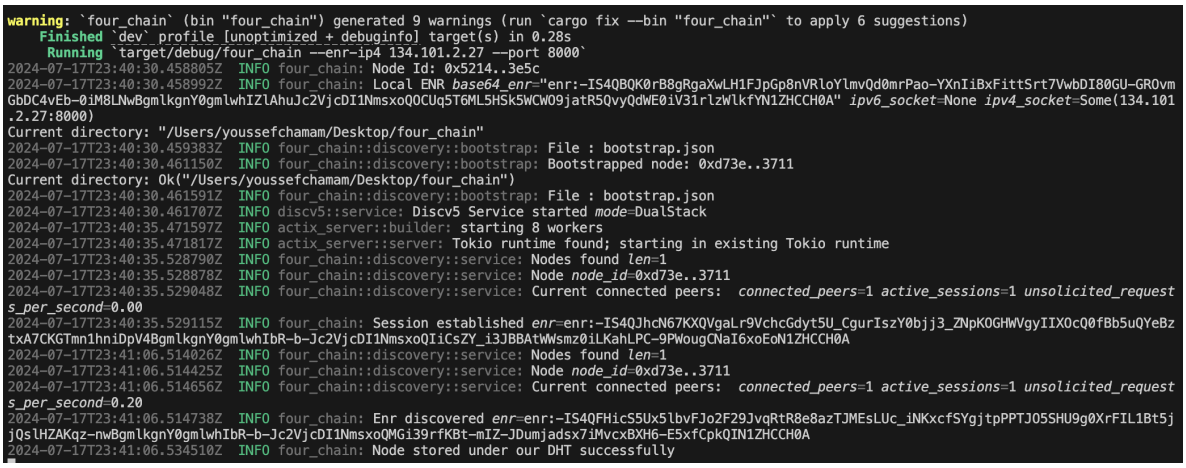


Figure 14: Node B

applications. It is particularly useful for handling numerous network connections. [11] We do this by making use of the tokio.spawn() method as follows:

```
tokio::spawn(run_discovery_loop( discv5, dht_protocol_for_loop));
```

Our run_discovery_loop is responsible for querying nodes on the networking level. It makes use of the Discv5 node lookup method described in section 2.3.4, which extends the Kademlia Protocol. As stated earlier, it performs a node lookup every 30 seconds as follows:

```
pub async fn lookup_nodes(discv5: &Discv5) {
// Initiate peer search
let target_random_node_id = enr::NodeId::random();
match discv5.find_node(target_random_node_id).await {
    Err(e) => {
        warn!(error = ?e, "Find Node result failed")
    }
    Ok(v) => {
        // found a list of ENR's print their NodeIds
        let node_ids = v.iter().map(|enr| enr.node_id()).collect::<Vec<_>>();
        info!(len = node_ids.len(), "Nodes found");
        for node_id in node_ids {
            info!(%node_id, "Node");
        }
    }
}
let metrics = discv5.metrics();
let connected_peers = discv5.connected_peers();
```

```
    info!(
        connected_peers,
        active_sessions = metrics.active_sessions,
        unsolicited_requests_per_second =
            format_args!("{:.2}", metrics.unsolicited_requests_per_second),
        "Current connected peers: "
    );
}
```

As we can see, the lookup nodes function takes the discv5 server as a parameter. It then starts with a random node ID that is generated with the help of the provided function from the discv5 library, that will then be narrowed down with the help of the XOR metric by calculating distances away from the node, performing the node lookup iteratively. It Runs an iterative FIND_NODE request that are sent through the secure connection already established between the known peers that have gone through the handshake process and are already stored under our local routing table with a secure established connection.
Responding nodes send back a NODES message containing ENR (Ethereum Node Records) of nodes that are closer to the target ID. The local node updates its routing table with the new nodes it discovers from the NODES message.
This will return peers containing contactable nodes of the discv5's DHT closest to the requested NodeId. Upon finding these nodes, the library will handle the new connections by establishing the handshake process and securing the new communication channels with the newly discovered peers. the protocol automatically initiates the handshake to establish or refresh the session. It also handles encryption and decryption of messages as well as the verification of node identities.

4.2.4 Discv5 Event Streams

Upon reading the discv5 library, one can see that it also includes the possibility of streaming the Events. It is therefore possible to create an event stream channel which can be polled to receive Discv5 events.
Within our discovery loop function, we are also capturing all of the events that are being sent by discv5 to further optimize our application. A few of these events that can be captured are:

- **Discovered** - This event as it says is captured when a node has been discovered. It returns the node' ENR.
- **NodeInserted** - Says that a node has been inserted into the routing table. Returns the node inserted ID, and if it replaced another node ID, it also returns it
- **SessionEstablished** - Handshake process has been established with a node from

our local routing table - Returns the ENR of the node with whom we have established the session

- **SocketUpdated** - Multiaddress of a node has been updated - Returns the new address in the form of the following struct :

```
pub struct SocketAddrV4 {
ip: Ipv4Addr,
port: u16,
}
```

- **TALKRequest** - Talkrequest received - Returns a TalkRequest struct composed of the following :

```
pub struct TalkRequest {
id: RequestId,
node_address: NodeAddress,
protocol: Vec<u8>,
body: Vec<u8>,
sender: Option<mpsc::UnboundedSender<HandlerIn>>,
}
```

Node Update

Next, nodes that are stored under our DHT routing table can have their ENRs updated. These ENRs might include new IP addresses / Ports, so we need to take this into consideration and perform updates accordingly. For those reasons, we can utilize the SocketUpdated event stream from Discv5 as follows:

```
Event::SocketUpdated(addr) => {
    info!(%addr, "Socket updated");
    //Find key in dht and update addr + port
    let ip = addr.ip().to_string();
    let port = addr.port();
    let node = Node::new(ip, port);
    interface.ping(node); //Pinging node to add it to dht
},
```

Code Explanation: This time, the stream provides a SocketAddr struct, so there is no need to derive any ENR. We simply derive the address and port, create a new Node instance and ping it. As usual, if the node responds to the PONG, the DHT Protocol will automatically store the new node information under our routing table.
Since we do not receive the ENR under this stream event, we are unable to directly eliminate the old node information from the DHT. However, as discussed in section 2.2.6, the Distributed Hash Table stores the node information in a queue that follows a

FIFO setup. This means that the least used addresses will find themselves at the end of the list, and will then be further replaced by this process and therefore eliminated from our buckets automatically.

4.2.5 TALK Request & TALK_RESP

This is a practical exploration of the TALK request from discv5 to showcase how it can be utilized, but it doesn't serve much purpose for now for the implemented code. However, for future work, the information being passed can serve for further optimization of our DHT querying system. As previously explained, upon receiving a new connection with a new node, we map its node ID to 0 in our routing table. To maintain this value, we are performing a TALK request to all the known peers periodically, passing to them in the body the number of known peers. The TALK request is then detected with the help of the event stream, and we parse the information received to update the data within our dht using the node ID as the key, which is received from the node_address, and the value is received from the body. This information exchange between the nodes can be split in two parts, the talk request, and the talk response.

TALK REQ

The TALK_REQ is sent periodically along with the rest of our loop function.
The Code :

```
let protocol = "peer_size".as_bytes();
//Finding all node Ids known to the current disc
let ids = discv5.table_entries_id();
for node in ids{
    //Finding known enr from the node ids
    let enr = discv5.find_enr(&node);
    //If enr is found, perform a talk request with it
    if let Some(found_enr) = enr {
        let _ = talk(&discv5, &interface, &found_enr, protocol).await;
    }
}
```

As we can see, we first start by defining the protocol over which we will be comunicating, which is a String transformed into bytes before transfer. This can serve as an identifier to the receiving node to know which data to pass / function to execute.
We are then using discv5 to call a method called table_entries_id(), which in turn returns a list of node IDs that are known the local node. We then loop through each of these ids to retrieve the latest known ENR we have in store on the routing table of discv5 associated to that node ID with the function find_enr, passing the ID to it. If we do have an ENR for the given node ID, because the field is optional in case we have none, we execute a talk request by performing our talk function.

The talk function takes the discv5 server as argument, the dht interface, the ENR that we have just found from the node ID and finally the protocol over which we will be executing the communication with the node. The talk function can be depicted below:

```
pub async fn talk(
    discv5: &Discv5,
    interface: &Protocol,
    enr: &enr::Enr<CombinedKey>,
    protocol: &[u8],
) -> Result<(), String> {
let peer_count = discv5.connected_peers().to_string();
let request_data = format!("{}", peer_count).as_bytes().to_vec();

// Send the talk request
match discv5
    .talk_req(enr.clone(), protocol.to_vec(), request_data)
    .await
{
    Ok(talk_response) => {
        // Process the talk response
        // Assuming the response is also in the form "node_id|peer_size"
        if let Ok(response_str) = std::str::from_utf8(&talk_response) {
            let parts: Vec<&str> = response_str.split('|').collect();
            if parts.len() == 2 {
                let remote_node_id = parts[0];
                let remote_peer_size = parts[1];
                interface
                 .put(remote_node_id.to_string(),remote_peer_size.to_string());
            }
        }
        Ok(())
    },
    Err(e) => Err(e.to_string()),
  }
}
```

Code explanation: we execute the asynchronous talk_req method from discv5 passing the receiving node's ENR, the protocol and the request data. We then match the response to what can be expected, which is whether an empty response or a talk_response. When it is a talk response, we are receiving the same information but for the receiving node. Meaning that if node A communicates over a talk request with node B to send how many nodes it can reach, node B will proceed by replying to node A with the same information from its side. After the talk request is completed and a talk response is

received, we will end up having both nodes information up to date under our DHT.

TALK RESP

In turn, the talk response is executed once a talk request is received. In order to achieve this, we are also making use of the event stream from discv5 here. We can read the following code to achieve this second part of the process :

```
Event::TalkRequest(talk_request) => {
    if talk_request.protocol() == "peer_size".as_bytes() {
        let request_body = talk_request.body();

        let known_peers_remote = match std::str::from_utf8(request_body) {
            Ok(v) => v,
            Err(_) => {
                let _ = talk_request.respond(vec![]);
                return;
            }
        };
        let node_id = talk_request.node_id().to_string();
        info!("talk request received from peer {}", node_id);
        // Storing known peer size to node ID
        interface.put(node_id, known_peers_remote.to_string());

        let known_peers = discv5.connected_peers();
        let self_id = discv5.local_enr().id();
        if let Some(enr_id) = self_id {
            let response = enr_id + "|" + &known_peers.to_string();
            let _ = talk_request.respond(response.as_bytes().to_vec());
        }
    }
}
```

Code Explanation: We receive the talk_request by capturing it from the event stream. It has one parameter which has all the information needed, including the ID of the node, and the body containing the information that we will be storing under the routing table, which is the number of peers that the sender node is able to reach. We define the node ID and the known peers number, that we then use to store under our DHT. We then proceed by retrieving the number of connected peers under our local discv5 using the method from the library discv5.local_enr().id(), and emit a response to the talk_request by passing the node ID and the known peers in a single string separated by ”—”, transformed to bytes.

Upon running the code and executing it on both the local computer and the droplet from digital ocean, we can see that the TalkRequest is being correctly received by the nodes:

Figure 15: Node A Receiving talk request

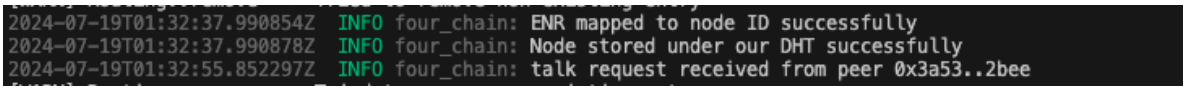


Figure 16: Node B: Receiving talk request

4.2.6 DHT Interface Integration with Discv5

Alongside our Discv5 server, we also go through the creating of our Distributed Hash Table interface, which will be responsible for the storage and retrieval of data by exposing its functionalities with a REST API that is reachable by anyone that knows the IP address / Port of any node in the system.
We integrate our DHT by first retrieving the bootstrap Node to pass it along to the DHT as well, by deriving the node information from the ENR to make sure we have a secure connection as follows:

```
 //Using bootstrap.json to get an optional Node
 //that we pass to our interface to bootstrap
let bootstrap_result = get_bootstrap_if_exists(bootstrap_file);
//Starting root with local ip address and port + 1
let root = Node::new(utils::get_local_ip().unwrap(), 8001);

//DHT interface responsible for adding nodes and data
let dht_protocol = Arc::new(Protocol::new(
    root.ip.clone(),
    root.port.clone(),
    bootstrap_result,
));
```

We now have our distributed hash table ready for usage, and it is bootstrapped with a secure and reachable node. In our application, we are relying on the capturing of these events to keep provide node information and updates to the Kademlia DHT as follows:

Node Insertion

```
Event::Discovered(enr) => {
    //Derive ip address and port from enr as well as node ID
    info!(%enr, "Enr discovered");
    //Pinging new discovered node to store it in our dht
    let enr_info = derive_info(&enr);
    if let Some(ip) = enr_info.udp4{
        let node = Node::new(ip.ip().to_string(), ip.port() + 1);
        //Storing the new node by pinging it
        let res = interface.ping(node);
        //Mapping the nodeId to the current known peer size
        //for later optimzation
        let id = derive_id_from_enr(&enr);
        if let Some(node_id) = id {
            //Initializing known peers to 0
            interface.put(node_id.to_string(), 0.to_string());
            info!("ENR mapped to node ID successfully");
        }
        if res{
            info!("Node stored under our DHT successfully");
        }else{
            info!("Failed at receiving PONG response. Node not stored.");
        }
    }else{
        info!("Could not derive node information")
    }
},
```

Code explanation: The discovered event stream captures the ENR of the node that has been newly discovered. This variable is passed to the internal function where we will derive the node information from it using the derive_info function. The derive info function is as follows:

```
    pub fn derive_info(enr: &enr::Enr<CombinedKey>) -> NodeInfo {
    let node_id = enr.node_id();
    let udp_port = enr.udp4_socket();

    NodeInfo {
        node_id: node_id,
```

```
        udp4: udp_port,
    }
}
```

Once our function returns the node ID and udp address containing both the IP address and the port, we create a Node struct using this derived information and ping it. If the ping returns a pong, the node is automatically handled and stored in our DHT local routing table. Furthermore, we initialize a value 0 to mapped to the node ID, representing the number of nodes that this peer has within the routing table of its ENR. This number is a form of initialization, and will later be modified using the TALK request, as presented in section 4.2.5.
We can derive the following sequence logic between nodes:

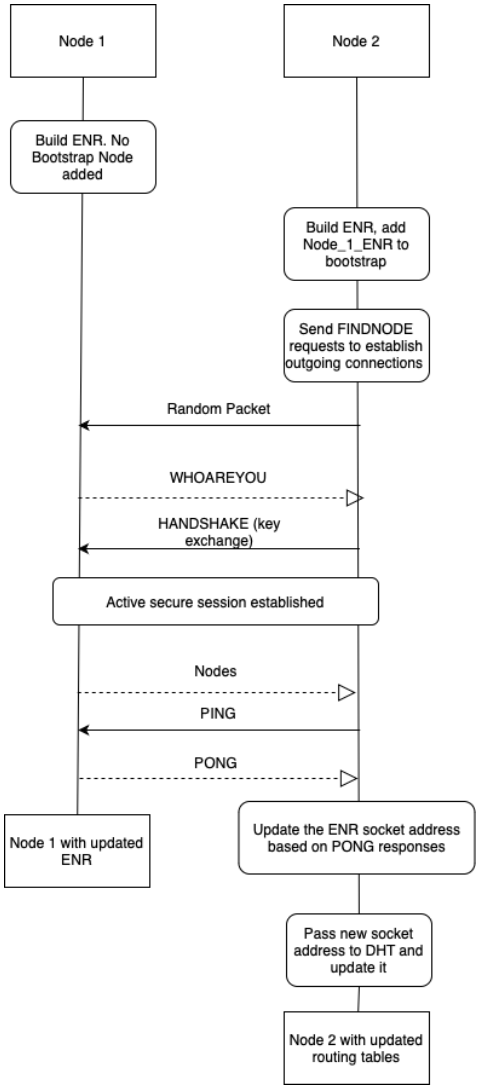


Figure 17: Process Flow for Node Discovery followed by Node insertion in DHT

4.3 Demonstration

Upon exposing our endpoints, we are now able to see if the DHT is truly storing and retrieving data.
This process will serve us to prove four key aspects:

- Our routes are operational and reachable.
- The Discv5 is operating as intended, discovering new nodes, generating ENRs from which we can derive node information that we can then use to communicate with.
- We are correctly deriving the node information and storing a new node instance under our DHT.
- We are able to store data with the DHT logic across the nodes, and this data can then be retrieved.

We can validate all these three points if we are able to make a request to our endpoints to store data, and then retrieving it.
For this, we will be using Postman, a platform for testing API endpoints, to make the API calls to a random node and see if the data is stored and then retrieved.

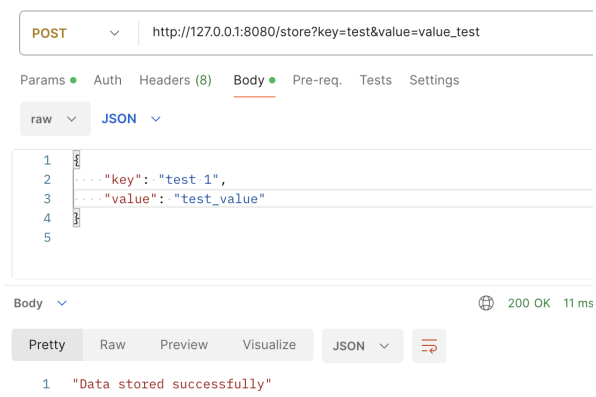


Figure 18: Store request

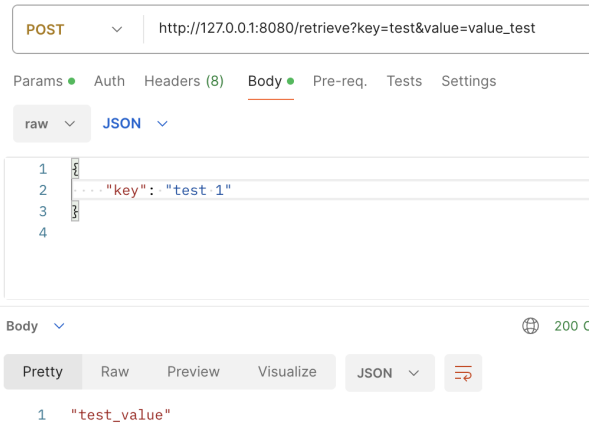


Figure 19: Retrieve request

As we can see in figure 18, we are making a call to our local machine which is running the first node instance, node A. We are making a call by passing the localhost Ip address 127.0.0.1, followed by the port that is used to create our web application. We are passing the data needed for the Store POST request, which is the key and the value, that are used to store the data in our DHT. Upon querying the key we just stored in figure 19, we successfully receive the value printed out.

We can wrap all of these functionalities within a sequence diagram to have a clearer understanding of how the process works:

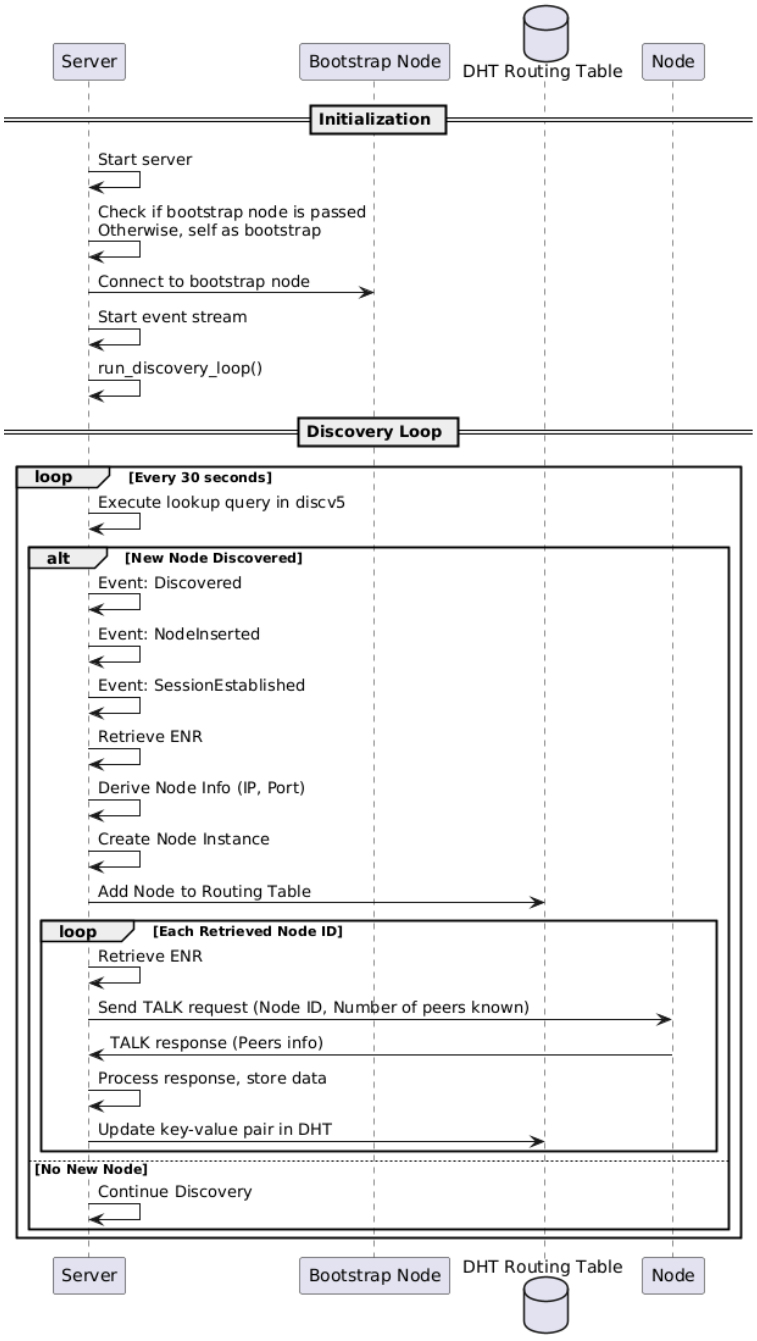


Figure 20: Sequence diagram of our application

5 Testing & Evaluation of Discv5 under different Scenarios

In this phase, we will evaluate specific discv5 scenarios that can occur to demonstrate its usability by taking into consideration potential cases that can happen and how it reacts depending on the given scenario. The testing in this paper is therefore solely reserved to Discv5 only. This part is inspired by a rigorous testing made by an open-source contributior found on Github, that is also contributing to the consensus mechanism of Ethereum. [3]

5.1 Tools Overview

In the development and evaluation of the Discv5 library, a variety of tools were employed to ensure rigorous testing. Below is a detailed overview of each tool used:

- **Testground** played a key role in assessing the network's performance at scale. It is a simulation platform specifically designed for running network simulations that can mimic large-scale network environments. Using Testground, the behavior of the network under various conditions, such as high traffic and node failures, was observed. This provided invaluable insights into the network's resilience and efficiency, guiding further optimizations and adjustments to ensure the network's stability and performance in production scenarios.

- **Docker** is an open-source tool designed to make it easier to create, deploy, and run applications by using containers. Containers allows us to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. Testground makes use of Docker containers to containerize test plans, which are the executable components of the Testground framework. [10]

5.2 Setting up our Testground Environment

To get Testground running, pre-requisites are having the Go language installed as well as having a docker daemon container running.
In order to run our testing plans, we need to run testgound within a docker container. For that, we start by running a docker-daemon container. We do so by installing the dependencies needed for testground within the repository using the following command:

```
make install
```

Once testground is properly installed, we can run the container with :

```
testground daemon
```

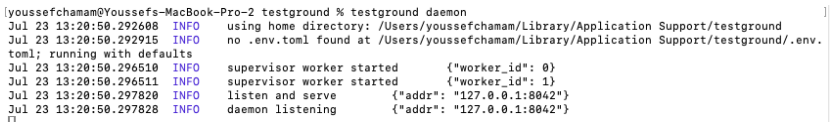


Figure 21: Testground running on Terminal

Figure 22: Overview of Docker Containers

We can now see the following appear on the terminal:
Finally, we can see the different docker containers running as follows:
Our Testground environment is now correctly set up and ready for the different test cases we will be going through to ensure the different scenarios that can occur and how discv5 handles these.
In the following scenarios, we will begin by presenting the test case, and the expected behaviour. We will then proceed by showcasing how to run the test. Once run, we will provide the details revealed and analyze them. We will finally conclude with the result of the test and how successful discv5 has been at handling the scenario, providing an evaluation.

5.3 Find Node

As proven already in section 4.2.3, find node is working correctly with 2 nodes. This test is meant to showcase how it is working with more peers in the system. We will be testing it with more than 2 servers running on testground to analyze how the nodes interact and find each other.
We have 4 nodes, and 1 boostrap node. Every node will attempt to find one another. The nodes begin by exchanging ENRs in order to use them and connect to each other:

Figure 23: Node information exchanged with the ENRs

As described in section 2.3.2, the ENR contains networking information as well as

Discv5 necessary information such as the Node ID and the signature as well as the public key of the generated keypair by the given node. Once the ENRs have been collected, the nodes will proceed to establish secure sessions between each other so that they can send Remote Call Procedures to each other.

Figure 24: Session establishment Process between Nodes

As we can see in figure 24, the peers are starting sessions between one another. We can see through the lines the process that they go through:

- Send a random packet to a Node
- Message received from other node without session
- Requesting back a WHOAREYOU packet to the requester
- WHOAREYOU Packet response received
- Sending back an Authentication response to the original recipient node
- Session established with node - Node promoted to connected

Following this sequence of packets being sent between the nodes, everyone of the nodes is able to be upgraded and added to a given node routing table. Once this session is established between the nodes, a findnode request can now be successfully sent by a peer and received by another in order to respond to it.
As we can see in figure 25, we have an iterative query event to find a given target. The lines begin by The FindNode request, that has been passed from node to another until the target has been found. We then calculate the distance between the target node that we have just found and the node that has made the query. Based on this distance, it would be allocated to the designated k-bucket in the routing table. The rest of the

Figure 25: FINDNODE Responses

requests display a node query for targets that already exist within the routing tables of the nodes, so they are automatically found without the need of contacting further neighbouring nodes. We can see the queried node information being printed in the terminal, derived from its ENR, including connectivity information.
Finally, this test has been accomplished and the test case is finished with a success, since all the nodes were able to find each other with the bootstrapping process, to then establishing secure connections and were able to exchange findnode requests in order to fill their routing tables.

5.4 IP Change

We will begin by stimulating two different nodes and performing an IP update on one of the nodes to see the changes in ENRs. As the flow chart in figure 26 describes it, this first test will cover two nodes, 1 & 2. One node will serve as the bootnode to the other. It will be added to the routing table, followed by a handshake process in order to establish a secure connection between them by exchanging the keypairs generated with their ENRs respectively. After that, one of the nodes (Node 1) will see its IP address being changed, but the ENR will not see any change. This should result in its update in the Node 2 routing table with a new ENR after a further PING/PONG is made between both nodes.

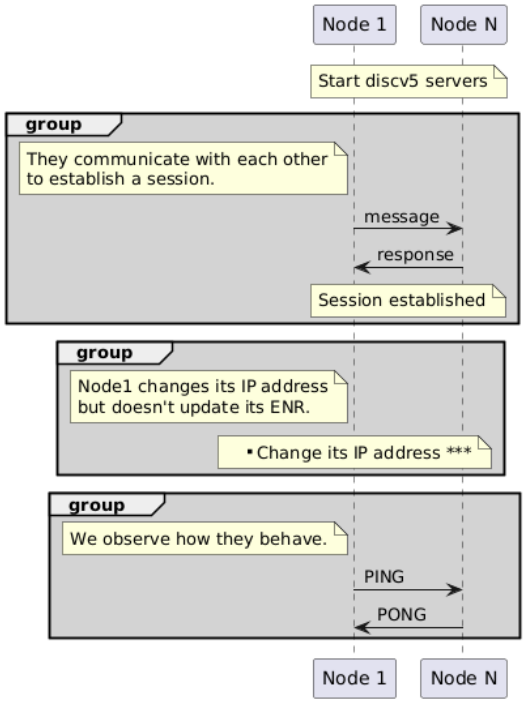


Figure 26: Process Flow for Node Discovery followed by an IP update

Upon running the test case, we can see the nodes going through the standard bootstrap process where they store the ENR and proceed with PING/PONG packets in order to then send authentication packets and to finally establish a secure connection between each other, over which they will communicate. In this figure 27, we have underlined the 5 major events that are occurring. We can rule them out as follows:

- External Ip address for node 002 has been modified. Previous address: 16.0.0.2. New address: 16.0.0.5.
- It proceeds and sends a PING request to node 001.
- As intended, the node 001 receives the message and detects that there is no session established between these nodes.Note that there has been a session establishment process. But the ENR currently held by node 001 is invalid. It has networking information over the previous ip address. Therefore, it doesn't recognize this node, and the packet received is a message over no secure session.
- Next, node 001 proceeds by sending a WHOAREYOU request back to node 002. The previous PING message has now timed out, and is therefore ruled out.
- Finally, the node 002 receives the WHOAREYOU packet and sends back an authentication response after it has generated the keypairs needed for the secure connection.

Figure 27: IP Change and Re-Authentication Process

We can finally see that the session with node 001 has been established in phase5. Below it as a side note, underlined in green, we can see that node 000 has been trying repetitively to reach node 002, but the requests have been failing with a time out since the node is not responding to them because of the IP change. On the other hand, we

Figure 28: Expired Session Handling Process

can see that node 000 has found node 001 through node 002 and is going through the same identification process, beginning with the WHOAREYOU request and so forth, until it will eventually secure a connection with the node.
This test case ends with a success. To summarize, nodes have made an initial connection. Then, one of the nodes has changed its external IP address without updating its local ENR, and proceeded by sending a ping request to another node. The recipient peer notices that the request happens over an unsecure connection, so responds back with a WHOAREYOU request. They end up establishing a connection, and all the

messages that were sent over an unsecure connection have been dropped by a time out. The node with the new ip address responds to the WHOAREYOU by sending an authentication packet, and a new session is established. The recipient node discovers that the ENR is invalid, and sends a one time session Ping message, to which a PONG message will be responded to, therefore determining the most recent Ethereum Node record maintained, and storing it under the routing table. The node then propagates its new ENR to the rest of the other nodes by going through this same process.

5.5 ENR Update

Much like the IP update, we have an initial session establishment between the nodes as we can see in figure 29. We now have our secure connections ready for messaging between the peers.

Figure 29: Session Establishment Between Nodes

Figure 30: Updating ENR & Emitting a PING Request

Node A proceeds and updates its local UDP socket contained in the key value pairs of its local ENR as we can see in 30.

Figure 31: Socket Updated Event

The node responds to ping requests from other peers with a new ENR, which results in a SocketUpdated Event, to the other peers that are connected to it. We can trace those messages to the rest of the nodes and see that the messages have been successfully received. The neighbouring peers have now the new ENR of the initial node.

```
>>> Result:

Jul 23 17:14:48.092903  INFO     finished run with ID: cqfu9v3juspkj70es9b0
Jul 23 17:14:48.093229  INFO     result default[cqfu9v3juspkj70es9b0]: success
youssefchamam@Youssefs-MacBook-Pro-2 testground % `
```

The test results in a success as well. The ENR update process went smoothly.

5.6 Concurrent Requests in Discv5

Having explored the foundational aspects of node interaction within the discv5 protocol, including dynamic IP address adaptation and ENR updates, we now turn our attention towards understanding how discv5 manages concurrent operations. This aspect is critical, as discv5 is designed to operate in highly dynamic and distributed network environments where multiple operations occur simultaneously.

5.6.1 General Concurrent Requests

In this test case, we want to see how a node typically reacts to several concurrent WHOAREYOU requests. We have 2 nodes, A and B. Node A will be executing parallel FIND_NODE requests to node B. Node B will be set with a short term session timeout in order to have an expired session (a few seconds), while receiving the requests. We can then demonstrate how discv5 handles the queue of incoming messages while making sure that they are only communicated over a secure connection.

Node B is serving as the bootstrap node . Therefore, node A has the ENR of node B. It proceeds by sending a random packet to node B. This packet is received by node B, which will go through the handshake process of establishing a secure connection with node A as depicted in 32.

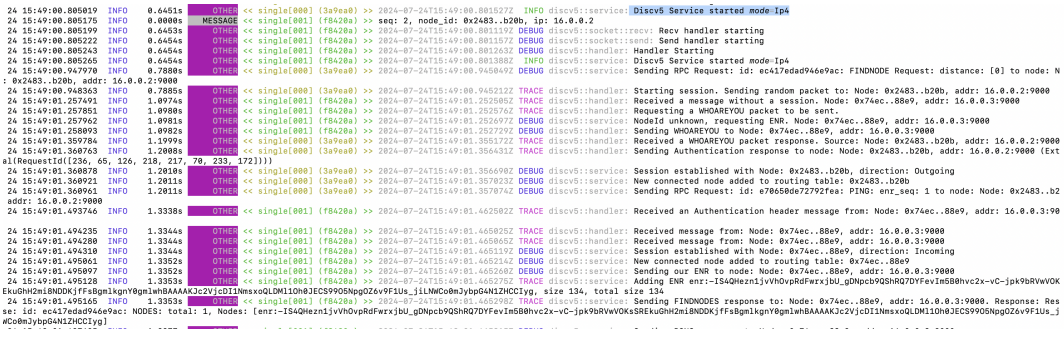


Figure 32: Session Establishment Between Node A & B

We can see that in the figure 33 that as soon as the new ENR has been added to our routing table and that there is a secure session between both nodes, the node B looks back at the received queries in order to start responding to them over this newly established secure connection.

```
 Session established with Node: 0x473c..ee5e, direction: Incoming
 New connected node added to routing table: 0x473c..ee5e
 Sending our ENR to node: Node: 0x473c..ee5e, addr: 16.0.0.2:9000
 Adding ENR enr:-IS4QEmIqkLJbsN4oBk2XCj0jZU7LV0sIJI8J_vyWxCfQ5fgI2_wSwWKwoELkq_h9ZQyx3
ize 134
 Sending FINDNODES response to: Node: 0x473c..ee5e, addr: 16.0.0.2:9000. Response: Res
AfdkFkOUnCp9EBTpSdWUABgmlkgnY0gmlwhBAAAAOJc2VjcDI1NmsxoQPbpEcR6Bif4yD4-w_Xe6reGiJP3jah

 Sending PONG response to Node: 0x473c..ee5e, addr: 16.0.0.2:9000
 Received message from: Node: 0xb24d..ebe2, addr: 16.0.0.3:9000
 Received message from: Node: 0xb24d..ebe2, addr: 16.0.0.3:9000
 Received RPC response: NODES: total: 1, Nodes: [enr:-IS4QEmIqkLJbsN4oBk2XCj0jZU7LV0sI
yzWX2ptkIN1ZHCCIyg] to request: FINDNODE Request: distance: [0] from: Node: 0xb24d..eb

 Received RPC response: PONG: Enr-seq: 1, Ip: 16.0.0.2,  Port: 9000 to request: PING:
```

Figure 33: 1st Response over Secure Connection

Since we set discv5 to have a short session timeout, we can see the following happening in figure 34:

1. Multiple parallel requests are being sent over from node A to node B.
2. Session times out. We can read that the messages are being received without a session.
3. Node B is sending multiple WHOAREYOU requests as a response to the FINDNODE parallel requests
4. It detects that a WHOAREYOU packet has already been sent, therefore aborts from the second packet.
5. we proceed with the Authentication. Once the Authentication packet is received and the node is added again to our routing table, Node B now sends the response to the FINDNODE requests.

The process runs logically in a sequential manner that mitigates any redundancy in the WHOAREYOU packets being sent, establishing a new connection again with the same node and ensuring the all requests are treated over a secure connection. The process ends with a success yet again.

```
Sending RPC Request: id: f204ef3dfe33ead4: FINDNODE Request: distance: [0] to node: N

Sending RPC Request: id: 19b9be7efd5cef94: FINDNODE Request: distance: [0] to node: N

Received a message without a session. Node: 0x473c..ee5e, addr: 16.0.0.2:9000
Requesting a WHOAREYOU packet to be sent.
Received a message without a session. Node: 0x473c..ee5e, addr: 16.0.0.2:9000
Requesting a WHOAREYOU packet to be sent.
Sending WHOAREYOU to Node: 0x473c..ee5e, addr: 16.0.0.2:9000
WHOAREYOU already sent. Node: 0x473c..ee5e, addr: 16.0.0.2:9000
Received a WHOAREYOU packet response. Source: Node: 0xb24d..ebe2, addr: 16.0.0.3:9000
Sending Authentication response to node: Node: 0xb24d..ebe2, addr: 16.0.0.3:9000 (Ext

Replaying active requests. Node: 0xb24d..ebe2, addr: 16.0.0.3:9000, Some([36, 155, 21

Session established with Node: 0xb24d..ebe2, direction: Outgoing
Received an Authentication header message from: Node: 0x473c..ee5e, addr: 16.0.0.2:90

Received message from: Node: 0x473c..ee5e, addr: 16.0.0.2:9000
Received message from: Node: 0x473c..ee5e, addr: 16.0.0.2:9000
Session established with Node: 0x473c..ee5e, direction: Incoming
Sending our ENR to node: Node: 0x473c..ee5e, addr: 16.0.0.2:9000
Adding ENR enr:-IS4QEmIqkLJbsN4oBk2XCj0jZU7LV0sIJI8J_vyWxCfQ5fgI2_wSwWKwoELkq_h9ZQyx3
ize 134
Sending FINDNODES response to: Node: 0x473c..ee5e, addr: 16.0.0.2:9000. Response: Res
ᵴfdkFkOUnCp9EBTpSdWUABgmlkgnY0gmlwhBAAAAOJc2VjcDI1NmsxoQPbpEcR6Bif4yD4-w_Xe6reGiJP3jah

Sending our ENR to node: Node: 0x473c..ee5e, addr: 16.0.0.2:9000
Adding ENR enr:-IS4QEmIqkLJbsN4oBk2XCj0jZU7LV0sIJI8J_vyWxCfQ5fgI2_wSwWKwoELkq_h9ZQyx3
ize 134
Sending FINDNODES response to: Node: 0x473c..ee5e, addr: 16.0.0.2:9000. Response: Res
ᵴfdkFkOUnCp9EBTpSdWUABgmlkgnY0gmlwhBAAAAOJc2VjcDI1NmsxoQPbpEcR6Bif4yD4-w_Xe6reGiJP3jah

Received message from: Node: 0xb24d..ebe2, addr: 16.0.0.3:9000
Received message from: Node: 0xb24d..ebe2, addr: 16.0.0.3:9000
Received RPC response: NODES: total: 1, Nodes: [enr:-IS4QEmIqkLJbsN4oBk2XCj0jZU7LV0sI
yzWX2ptkIN1ZHCCIyg] to request: FINDNODE Request: distance: [0] from: Node: 0xb24d..eb

Received RPC response: NODES: total: 1, Nodes: [enr:-IS4QEmIqkLJbsN4oBk2XCj0jZU7LV0sI
yzWX2ptkIN1ZHCCIyg] to request: FINDNODE Request: distance: [0] from: Node: 0xb24d..eb
```

Figure 34: Session Timeout - Managing Requests

5.6.2 Concurrent Requests before establishing Secure Session

In this test case, we will be sending TALK requests in parallel from one node to another, before there even is a secure connection between the node.

```
Jul 23 17:58:08.096990  INFO    0.7662s       OTHER << single[000] (fbd055) >> 2024-07-23T17:58:08.094529Z  INFO discv5::service: Discv5 Service started mode=Ip4
Jul 23 17:58:08.097037  INFO    0.7662s       OTHER << single[000] (fbd055) >> 2024-07-23T17:58:08.094569Z DEBUG discv5::handler: Handler Starting
Jul 23 17:58:08.097080  INFO    0.7663s       OTHER << single[001] (e4d88f) >> 2024-07-23T17:58:08.094555Z  INFO discv5::service: Discv5 Service started mode=Ip4
Jul 23 17:58:08.097282  INFO    0.7665s       OTHER << single[001] (e4d88f) >> 2024-07-23T17:58:08.094466Z DEBUG discv5::socket::send: Send handler starting
Jul 23 17:58:08.097309  INFO    0.7665s       OTHER << single[001] (e4d88f) >> 2024-07-23T17:58:08.094577Z DEBUG discv5::handler: Handler Starting
Jul 23 17:58:08.138930  INFO    0.8081s       OTHER << single[000] (fbd055) >> 2024-07-23T17:58:08.136610Z DEBUG discv5::service: Sending RPC Request: id: bed2514f746d17fd: TAL
K: protocol: 00, request: 00 to node: Node: 0x025f..3e05, addr: 16.0.0.3:9000
Jul 23 17:58:08.139074  INFO    0.8083s       OTHER << single[000] (fbd055) >> 2024-07-23T17:58:08.136654Z DEBUG discv5::service: Sending RPC Request: id: d80f504fa8ef457b: TAL
K: protocol: 00, request: 01 to node: Node: 0x025f..3e05, addr: 16.0.0.3:9000
```

Figure 35: 2 TALK Requests Prior to Handshake

We can see in this figure 35 that 2 TALK requests have been sent as soon as the discv5 services started across the 2 nodes.

Figure 36: Handshake Process Establishment

The sending node notices that there is no secure connection established with the recipient node, and proceeds by sending a random packet to establish this secure connection with it. We can see a traced message by the sender peer saying that the request has been queued. We can see that one message has been placed under "active requests", and the rest of the messages (2 in total) are placed into "pending requests" locally.
It so proceeds and establishes the handshake process with the recipient node, exchanging authentication data as usual, and the node is now connected and officially added to the routing table in a designated k-bucket.
We now delve into the last step of this test:

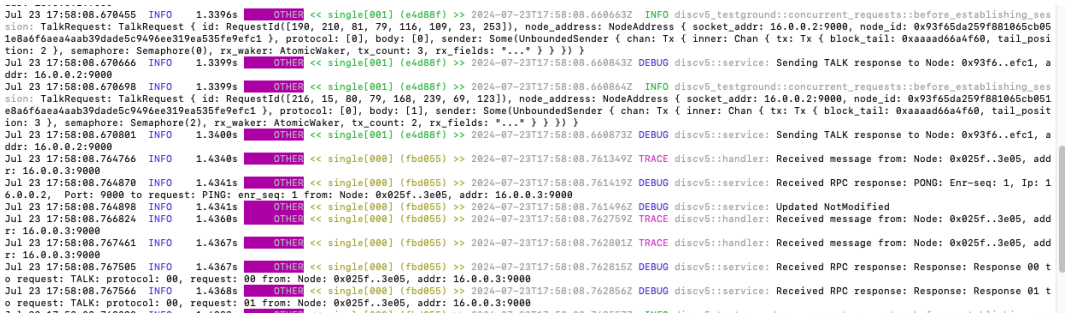


Figure 37: TALK Responses

As we can see, the sender node has the previous events received stored and fetches them back in order to re-send them over this newly established secure connection with the node 2, 1 by 1, by sending the queued TALK requests to the designated node. The messages are then received by the node in question, and the test results in a successful management of concurrent requests with no session establishment, by caching the requests, establishing a secure connection with the node and then going back to the requests in order to finally receive a response to them.

5.7 Eclipse Attack

Having explored the dynamic behavior of discv5 in handling IP and ENR updates as well as managing concurrency in distributed environments, we now shift our focus towards the security implications of these network dynamics.
Eclipse attacks represent a critical vulnerability in peer-to-peer networks as seen in section 2.3.5, where malicious nodes manipulate network connections to isolate a victim from honest peers. This type of attack is particularly concerning because it directly undermines the trust and reliability of network routing mechanisms. By monopolizing the victim's routing table with attacker-controlled entries, an adversary can effectively sever the victim's connections to the rest of the network, leading to misinformation or denial of access.
In the beginning, we see a list of multiple nodes, with on the one hand the victim node, and on the other hand an attacker node along with several malicious nodes that will come in hand-in-hand in the attack process with the attacker. These attackers will try to fill the routing table of the victim node in order to isolate it from the rest of the network. We finally have an honest node, which we will be using to check once the attack has been made, whether or not it is still able to reach our victim node. We can see the following list being printed out:

Figure 38: Eclipse Attack Scenario Nodes List

We then can attest that all the nodes are filling the routing table of the victim node by connecting to it as follows:

Figure 39: Attacking Nodes Information & Session Establishment

As we can witness, all the attackers are being inserted into our routing table. The attacker node has successfully established a secure connection with the victim node upon making a handshake process. Since both are now connected, the victim node was vulnerable to the attacker's incoming connection and has discovered the other accomplice nodes in the attack by detecting them with the Discovered Event from discv5 and then inserting them as a consequence.
Finally, we can print out the routing table of the victim node:

```
Jul 23 17:31:48.142787  INFO    0.0000s    MESSAGE << victim[000] (32e5a0) >> [KBucket] index:252, num_entries:0, num_connected:0, num_disconnected:0
Jul 23 17:31:48.142798  INFO    0.0000s    MESSAGE << victim[000] (32e5a0) >> [KBucket] index:253, num_entries:0, num_connected:0, num_disconnected:0
Jul 23 17:31:48.142812  INFO    0.0000s    MESSAGE << victim[000] (32e5a0) >> [KBucket] index:254, num_entries:0, num_connected:0, num_disconnected:0
Jul 23 17:31:48.142833  INFO    0.0000s    MESSAGE << victim[000] (32e5a0) >> [KBucket] index:255, num_entries:8, num_connected:8, num_disconnected:0
```

Figure 40: List of k-buckets from our Victim Node

We can read in figure 40 the following information:

- index - Index of our k-bucket from the routing table
- num_entries - Number of node entries in the given k-bucket
- num_connected - Number of currently connected nodes
- num_disconnected - Number of disconnected nodes

In figure 40, we have the last buckets from our routing table. We are specifically interested in the last one. Upon receiving the state of the k-buckets, we witnessed that all of them have 0 entries, except of the last k-bucket of index 256, which has 8 entries, all nodes being currently connected. This is due to the initial settings the nodes have been created with, which is a limit to the number of nodes per buckets, set to 8.
We can conclude that the attackers have successfully filled that last 8-bucket, but have dramatically failed at taking over the entire routing table with this parameter being set. The attack is therefore unsuccessful, and the victim node has not been isolated from the network. The honest node can still connect to the victim node since there are still empty 8-buckets in its routing table.
However, if we were to remove the parameter, the victim node emits "Table full" error and the test case results in failure, since the victim's routing table is full of the "incoming" attacker node ids.

5.8 Results Evaluation

Upon reviewing and analyzing each test case, we have covered potential scenarios that can occur when networking with discv5, and the results have been really well handled. We started by covering basic scenarios of communication between nodes when faced with changes of IP address with an invalid ENR, to then covering the potential concurrent request cases. Finally, we assessed how discv5 reacts to a major security attack that can be made to distributed systems, the eclipse attack.

Upon running the first test cases, we notice that nodes make sure that other peers always have the latest ENR. Nodes detect that an ENR is invalid when there is a different IP address as well, and make sure that they request a valid ENR, that the peer would generate. This ENR will then get propagated to ensure that it always has the latest valid one. Furthermore, sending requests to a node with no established session are not handled directly. They are cached within a queue of messages that are pending, and only treated once a handshake process is made between the nodes. Finally, the networking system responds successfully to the eclipse attack, where a malicious node tries to fill the routing table of another victim node. It fails in doing so by adding a parameter to set a limit of peers per bucket within the routing table, which would result in only one bucket being filled by malicious nodes, and the victim node is still able to handle upcoming requests from other honest peers.

6 Outline

6.1 Conclusion

In this thesis, we embarked on a practical exploration of the discv5 protocol, an essential component in the architecture of distributed systems. This investigation began with a theoretical overview of distributed systems, providing a foundational understanding necessary for delving into more complex concepts. We further examined the intricacies of discv5 and Distributed Hash Tables (DHT), focusing on their critical role in enabling peers within a network to discover and connect with one another efficiently.

Through the implementation of a system utilizing both DHT and discv5, this thesis demonstrated the operational capabilities of these technologies in creating and maintaining a structured pure p2p network. Notably, the use of discv5's routing tables and Remote Procedure Calls facilitated effective node discovery on the networking layer in a decentralized manner, underscoring the protocol's utility in real-world applications with a focus on having no central entity. Matching this technology with the distribution of data across the system of nodes with Kademlia Distributed Hash Tables goes hand-in-hand in order to fulfill those needs.

Our empirical analysis included a series of tests designed to challenge the protocol under various conditions, such as IP address changes without ENR updates, ENR updates themselves, and handling concurrent requests both before and after the establishment of a secure session. Additionally, the simulation of an eclipse attack, which was mitigated by the protocol's inherent limitations on node numbers per bucket, further highlighted the resilience of discv5 under potential security threats.

The findings from these tests confirmed that discv5 is not only theoretically sound but also robust in practical applications, capable of handling dynamic network changes and security challenges efficiently. However, several functionalities are promised theoretically, but are yet to be implemented practically. These would be interesting for an establishment of sub-protocols within the networking between peers.

6.2 Future Work

While this thesis has made significant contributions to understanding and applying the discv5 protocol in distributed systems, several promising areas remain unexplored, presenting opportunities for future research. One of the most notable areas involves the practical implementation of the topic advertisement mechanism, a theoretical aspect of discv5 that has yet to be fully realized in practical scenarios. Some of these can vary into the following:

- Implementation of Topic Advertisement Mechanisms
- Clustering Network Based on Topics
- Scalability and Performance Optimization with topic-based clustering
- Security Implications of Topic-Based Clustering
- Integration with Other Protocols

By pursuing these areas of future work, researchers and developers can continue to refine and expand the capabilities of discv5, potentially transforming it into a more versatile tool for managing distributed networks. This not only supports the growth and efficiency of existing systems but also opens up new possibilities for innovative and decentralized social media platforms.

References

[1] Big o notation.
URL: https://en.wikipedia.org/wiki/Big_O_notation.

[2] Block cipher mode of operation.
URL: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.

[3] Discv5 testing plans github repository.
URL:https://github.com/ackintosh/discv5-testground.

[4] Distributed hash tables wikipedia definition.
URL:https://en.wikipedia.org/wiki/Distributed_hash_table.

[5] Ethereum bootstrap nodes.
URL:https://github.com/ethereum/go-ethereum/blob/master/params/bootnodes.go#L23.

[6] Overlay network.
URL:https://en.wikipedia.org/wiki/Overlay_network.

[7] Rust crate official documentation.
URL:https://docs.rs/discv5/latest/discv5/#:~:text=Discovery%20v5%20is%20a%20protocol,optionally%20IP%20address%20and%20port.

[8] Serde - framework official homepage.
URL: https://serde.rs.

[9] Struct discv5::discv5.
URL: https://docs.rs/discv5/latest/discv5/struct.Discv5.html.

[10] Testground github repository.
URL:https://github.com/search?q=discv5&type=repositories.

[11] Tokio - framework official homepage.
URL: https://tokio.rs.

[12] What is a cryptographic nonce?
URL: https://www.okta.com/identity-101/nonce/#:~:text=Nonce%20in%20cryptography%20means%20œnumber,it%20is%20only%20used%20once.

[13] What is the osi model.
URL: https://www.imperva.com/learn/application-security/osi-model/.

[14] D. Acharya. *Resource Discovery in Distributed Systems*, 2022.
URL:https://www.researchgate.net/publication/363431109_Resource_Discovery_in_Distributed_Systems.

[15] I. Baumgart and S. Mies. *A praticable Approach Towards Secure Key-Based Routing*, 2007.
URL:https://attachment.victorlampcdn.com/article/content/20220705/SKademlia_2007.pdf.

[16] I. Baumgart and S. Mies. *A brief overview of Kademlia and its use in various decentralized platforms*, 2019. https://www.storj.io/blog/a-brief-overview-of-kademlia-and-its-use-in-various-decentralized-platforms.

[17] R. G. Bengt Carlsson. The rise and fall of napster – an evolutionary approach.
URL: https://kk.rs/sN5S1.

[18] A. Binzenhöfer and H. Schnabel. *Improving the Performance and Robustness of Kademlia-based Overlay Networks*, 2007.
URL:https://link.springer.com/chapter/10.1007/978-3-540-69962-0_2.

[19] Dean. *From Kademlia to Discv5*, 2020.
URL:https://vac.dev/rlog/kademlia-to-discv5/.

[20] *Evolving devp2p*, 2018.
URL:https://www.youtube.com/watch?v=YqGncHRtshM.

[21] Ethereum. Ethereum 2.0 networking specification.
URL: https://github.com/ethereum/consensus-specs/blob/065b4ef856aeb7f84f1bed5c4a2cd4d6ac1edc87/specs/phase0/p2p-interface.md#the-discovery-domain-discv5.

[22] Ethereum. Ethereum node records.
URL: https://github.com/ethereum/devp2p/blob/master/enr.md.

[23] Ethereum. Node discovery protocol v5 generality.
URL:https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md, 2020.

[24] Ethereum. Node discovery protocol v5 rationale.
URL:https://github.com/ethereum/devp2p/blob/master/discv5/discv5-rationale.md, 2020.

[25] Ethereum. Node discovery protocol v5 theory.
URL:https://github.com/ethereum/devp2p/blob/master/discv5/discv5-theory.md, 2020.

[26] Ethereum. Node discovery protocol v5 wire protocol.
URL:https://github.com/ethereum/devp2p/blob/master/discv5/discv5-wire.md, 2020.

[27] Ethereum. *Node Discovery Protocol*, 2023.
URL:https://github.com/ethereum/devp2p/blob/master/discv4.md#known-issues-in-the-current-version.

[28] F. Lange. Eip-778: Ethereum node records.
URL:https://eips.ethereum.org/EIPS/eip-778, 2017.

[29] F. Lange. Recursive-length prefix (rlp) serialization.
URL:https://ethereum.org/en/developers/docs/data-structures-and-encoding/rlp/, 2024.

[30] P. Maymounkov and D. Mazières. *A Peer-to-peer Information System Based on the XOR Metric*, 2002.
URL:https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf.

[31] Piertwo. Discv5 explainer series 1.
URL:https://piertwo.com/blog/discv5-explainer-series-2/, 2023.

[32] L. Strygul. *Kademlia: the P2P System Behind Ethereum and BitTorrent Networks*, 2021.
URL:https://medium.com/@ievstrygul/kademlia-the-p2p-system-behind-ethereum-and-bittorrent-networks-a3d8f539f114.

[33] A. S. Tanenbaum and M. V. Steen. *Distributed Systems - Principles and Paradigms*. Upper Saddle River, 2016.
URL:https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28Gschka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf.

[34] M. van Steen Andrew S. Tanenbaum. *A brief introduction to distributed systems*, 2016.
URL:https://link.springer.com/article/10.1007/S00607-016-0508-7.

[35] B. P. K. B. S. Vassiliadis. *A Survey of Peer-to-Peer Networks*, 2005.
URL:https://profile.iiita.ac.in/bibhas.ghoshal/lecture_slides/distributed/P2P%20Network%20Survey.pdf.

List of Image Sources

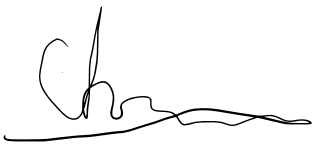
1. Centralized, Decentralized & Distributed Networks
   https://www.rand.org/content/dam/rand/pubs/research_memoranda/2006/RM3420.pdf
2. Layers on the OSI Model
   https://www.cloudflare.com/fr-fr/learning/ddos/glossary/open-systems-interconnection-model-osi/
3. Computer Systems,
   https://profile.iiita.ac.in/bibhas.ghoshal/lecture_slides/distributed/P2P%20Network%20Survey.pdf
4. Centralized Indexing,
   https://profile.iiita.ac.in/bibhas.ghoshal/lecture_slides/distributed/P2P%20Network%20Survey.pdf
5. Kademlia Binary Tree representing K-Buckets,
   https://www.youtube.com/watch?v=1QdKhNpsj8M
6. AES/CTR Encryption
   https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

List of Figures

List of Tables

Abbreviations

I hereby declare that this thesis titled, "An Examination of the discv5 Discovery Protocol in Distributed Applications," is entirely my own work. No part of this thesis has been submitted for any other degree or professional qualification. All sources and resources utilized in the preparation of this document have been properly acknowledged. I have adhered to all academic guidelines and ethical considerations throughout my research.

Berlin, 11.08.24, Signature: 
Ort, Datum, Unterschrift