POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Introducing Automatic Discovery Mechanisms for the Computing Continuum

Supervisors

Prof. Fulvio RISSO

Ing. Jacopo MARINO

Candidate

Matteo CICILLONI

ACADEMIC-YEAR 2024-2025

Abstract

The optimization of computing resources at the edge, combined with cloud-based computational power, has gained significant attention in recent years under the concept of the computing continuum. Within this context, the Flexible, scaLable, secUre, and decentralIseD Operating System (FLUIDOS) project has emerged as an implementation of this paradigm, enabling seamless resource management across edge and cloud environments. Among its potential applications are optimizing resource utilization for a fleet of robots, reducing power consumption while enabling intensive computations, and facilitating interaction between connected vehicles and infrastructure. A challenge in this landscape is the discovery of available computational resources.

The scope of this thesis is the design and implementation of an evaluation framework to benchmark, analyze and compare different discovery protocols within the FLUIDOS ecosystem. The framework provides a structured approach by defining network condition requirements, setting up a testbed with defined and repeatable conditions, and implementing a benchmarking tool compatible with the FLUIDOS architecture, capable of automating tests to ensure statistical significance. The goal is to compare various protocol alternatives based on meaningful performance metrics such as discovery time across varying network conditions and protocol behavior at scale.

Experimental results highlight the distinct behaviors of discovery protocols based on multicast and distributed hash table (DHT) overlay networks. The analysis reveals significant differences in discovery time, scalability, and inherent networking limitations, providing insights into the trade-offs between these approaches. These findings contribute to a deeper understanding of resource discovery mechanisms and offer a foundation for further development of discovery protocols in FLUIDOS, giving users the ability to choose the most suitable protocol for their specific use case.

Table of Contents

List of Tables

List of Figures

Acronyms

DS
Distributed System

VM
Virtual Machine

CRDT
Conflict-free Replicated Data Type

SEC
Strong Eventual Consistency

LWW Set
Last-Write-Wins Set

DHT
Distributed Hash Table

DDBs
Distributed Databases

DB
Database

RF
Replication Factor

IoT
Internet of Things

PoW
: Proof of Work

PoS
: Proof of Stake

PBFT
: Practical Byzantine Fault Tolerance

OSI
: Open Systems Interconnection

IP
: Internet Protocol

IGMP
: Internet Group Management Protocol

PIM
: Protocol Independent Multicast

UGV
: Unmanned Ground Vehicle

UAV
: Unmanned Aerial Vehicle

GPU
: Graphics Processing Unit

CPU
: Central Processing Unit

RAM
: Random Access Memory

ASIC
: Application-Specific Integrated Circuit

DLT
: Distributed Ledger Technology

Chapter 1

Introduction

In recent years, cloud computing has become a widely used approach for accessing computing power. It offers flexible and cost-effective solutions by allowing users to run applications on hardware managed by cloud providers. These providers operate large data centers and allocate resources based on different pricing models, such as pay-per-use. However, this centralized model does not take full advantage of the many computing resources available at the edge of the network, for examples small servers, Internet of Things (IoT) devices, and 5G base stations. Using these resources more efficiently could improve system performance, reduce delays, and achieve lower energy consumption. The computing continuum aims to address this challenge by integrating these resources into a more dynamic and adaptable system.

The computing continuum is based on the idea that computation should take place where it is most efficient. Tasks can be executed on small devices at the edge, on intermediate nodes, or in powerful cloud data centers, depending on factors such as latency, energy use, network conditions, and workload requirements. This flexible system allows data to be processed closer to where it is generated when possible, while still relying on cloud resources for more demanding tasks.

Within this context, the Flexible, scaLable, secUre, and decentralIseD Operating System (FLUIDOS) project has been developed with the aim to make resource management between edge and cloud environments more seamless and efficient.

An use case of FLUIDOS considered in this work is the management of computing resources for fleets of autonomous robots. Since these robots run on battery power and have not so powerful hardware, so limited in both operational time and available resources, it could be beneficial to offload computation to more powerful resources which have stable supply of energy, thus enabling the robots to run for longer periods of time and perform more intense computations that could not be possible with their onboard hardware. Another possible use case could be enabling better

communication between connected vehicles and infrastructure.

In all these cases, one important step is to ensure that devices can find available computing resources when needed. This is done through discovery protocols, which allow computing nodes to locate and communicate with each other. Since different protocols offer different advantages and limitations, it is important to evaluate their performance under various conditions.

1.1 Objectives

The main goal of this thesis is to design and implement a framework to test, analyze, and compare different discovery protocols within the FLUIDOS ecosystem. Having a structured and automated way to evaluate these protocols is useful for different reasons. First, it helps users choose the most suitable protocol for their needs by providing clear information about performance and trade-offs. Second, it serves as a tool for improving discovery protocols in FLUIDOS by offering a way to test new implementations and modifications.

The evaluation framework will be designed to interact with the FLUIDOS node, which represents an entity capable of computation within the FLUIDOS network. It will measure key performance factors such as the time it takes to find and connect to other FLUIDOS nodes under different network conditions and how well the protocol works at scale.

To ensure that the results are reliable and can be reproduced, the system will include automated testing under standardized conditions. This will allow for a fair comparison of different discovery protocols and help identify their strengths and weaknesses while ensuring statistical significance in the results.

1.2 Outline

The thesis is structured as follows:

- **Chapter 2**: analysis of modern data distribution technologies in distributed architectures, assessing whether established protocols utilize the most effective foundational technologies available

- **Chapter 3**: introduction to Kubernetes orchestration system, its architecture and relevant components

- **Chapter 4**: introduction to FLUIDOS, its architecture and components, and the discovery protocols currently used

- **Chapter 5**: implementation of the evaluation framework, its requirements, the design choices adopted and the implementation details of the benchmarking tool
- **Chapter 6**: results of the experiments and analysis of the data collected
- **Chapter 7**: conclusions and future work

Chapter 2

Data Sharing Technologies in Distributed Systems

The use cases described in Chapter 1, and computing continuum in general, can be abstracted as nodes in distributed system of computing resources across edge and cloud environments.

In this chapter distributed systems will be introduced together with their characteristics and challenges. Then, the state of the art for different technologies used to distribute data within the distributed systems context will be analyzed and finally they will be compared among themselves to identify the most suitable for the use cases described previously.

2.1 Distributed Systems

As computational demand grows for modern applications, scalability becomes an issue for centralized systems which can only scale vertically by adding more resources such as CPU, RAM and faster disks. This can become impractical: adding physical hardware needs physical intervention and there is a technological limit to the computational power that can be added to a single machine.

Furthermore, the problem of geographical distribution of the services provided by said applications becomes apparent: how can a centralized system provide services to users worlwide, especially considering the delay in response time due to physical distance?

These new problems call for a new paradigm: Distributed Systems (DSs). DSs can then be defined as computing systems in which multiple independent computing entities collaborate to solve complex computational tasks.

Unlike traditional centralized systems, DSs divide computational workloads

across numerous nodes across network, enabling improved scalability, reliability due to fault tolerance, and performance. Compared to centralized systems, distributed systems can scale horizontally by adding more machines to the network, usually via virtualization by dividing bare metal computing resources into virtual machines(VMs) or containers. As nodes are part of a network, they do not need to be in the same physical machine, but they can be located in different parts of the world. This becomes very valuable when considering the need for scalable applications that can be accessed by users around the world: for example, we can imagine an infrastructure with servers located in Europe, the US and Asia. This allows for more flexibility and resiliency for the whole infrastructure and is at the base of the modern cloud computing paradigm.

Naturally this approach does not come only with advantages, but also with new challenges to be solved. The fundamental challenge in distributed systems lies in coordinating multiple independent computational entities that operate concurrently, potentially across different physical locations and with possibility of communication failure across the network. These systems must address several challenges:

- Making sure data is consistent across all nodes
- Handling potential node failures without system-wide disruption
- Maintaining communication reliable across potentially unreliable network infrastructures
- Managing simultaneous process execution across multiple nodes

2.2 The CAP Theorem

The theoretical foundation for understanding distributed system limitations is the Consistency, Availability, Partition Tolerance (CAP) theorem. This theorem describes a fundamental trade-off in distributed system design: a system can simultaneously provide at most two of three following properties:

1. Consistency: All nodes observe identical data at the same time
2. Availability: Every request receives a timely response
3. Partition Tolerance: The system continues functioning despite faults in network communication

Notice that the CAP theorem is not an absolute rule equivalent to a mathematical theorem [1], but rather a guide line to answer the following question: which property

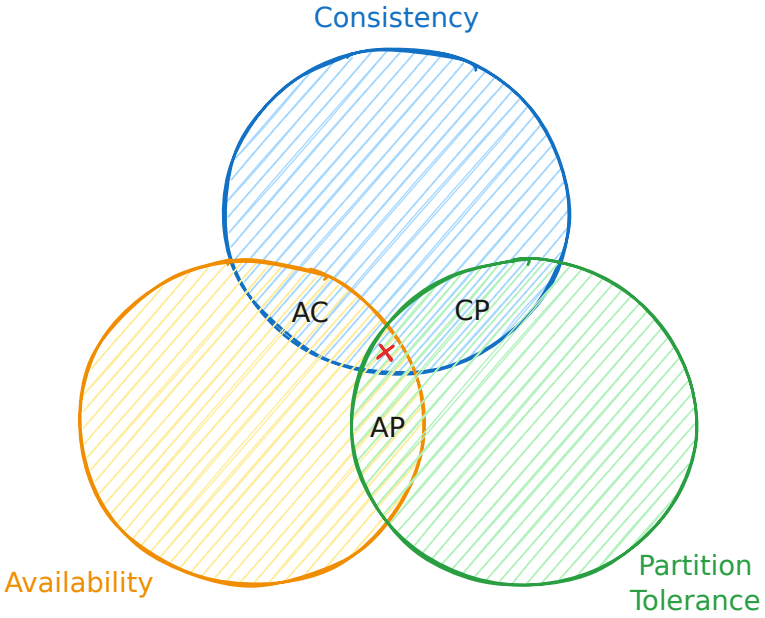

Figure 2.1: Venn diagram representing the intersection of the 3 CAP properties. The red area represents the impossible combination of all 3 properties

between Consistency and Availability is desired in a particular distributed system when a network partition occurs?

For the purposes of this research, given the inherent unpredictability of network communications, partition tolerance becomes a critical requirement. Since the use cases analyzed in the research problem are relative to physical moving systems, such as drones and connected vehicles, high availability is the characteristic that must be guaranteed.

The problem of discovering entities within a network can be simplified to the distribution of data, in our case at the bare minimum the knowledge of the ip address of another cluster.

The subsequent sections will explore various approaches to distributed data examining distributed databases, conflict-free replicated data types, distributed hash tables, blockchain technologies and multicast analyzing their advantages and disadvantages in order to compare them.

2.3 Conflict-Free Replicated Data Types

As discussed in section 2.2 in a highly available system strict consistency is sacrificed and it is possible that information that is not up to date is served to the user. What would happen in case of a concurrent update? How can conflicts be resolved? Conflict-Free Replicated Data Types (CRDTs) are a class of data structures that provide a solution to this problem since they are designed to be updated without the need for synchronization[2].

The main feature of CRDTs is that they are designed to be commutative and associative, meaning that the end result is independent of the order in which operations are applied. This is why they are strongly eventual consistent (SEC). SEC is a weaker form of consistency which guarantees - in absence of further updates - that all replicas will eventually converge to the same state without any form of synchronization.

The core principle behind CRDTs is the use of mathematically defined merge functions or commutative operations that ensure consistency across replicas. CRDTs are broadly classified into two categories:

- **State-based CRDTs:** These rely on merging states across replicas, where updates are propagated by exchanging the full state. They have higher communication overhead due to the full state transmission but are simpler to implement.
- **Operation-based CRDTs:** These propagate individual operations instead of the full state, relying on the commutativity of operations to ensure consistency. Not all operations can be commutative, for example in sets, in which case metadata containing ordering information is required to resolve conflicts.

Two examples to further understand the concept of state-based and operation-based CRDTs will be now presented.

2.3.1 State-Based CRDT Counters (CvRDT)

In a state-based counter each replica maintains a state vector where each entry corresponds to a replica's contribution. Updates modify the local state, and replicas periodically exchange and merge their states.

Let $R = \{R_1, R_2, \ldots, R_n\}$. Each replica R_i maintains a vector $C_i = [c_{i1}, c_{i2}, \ldots, c_{in}]$, where c_{ij} represents the count contributed by replica R_j as known by R_i. Local updates modify c_{ii}, the entry corresponding to the local replica:

$$c_{ii} \leftarrow c_{ii} + k \quad (k \in \mathbb{Z}^+)$$

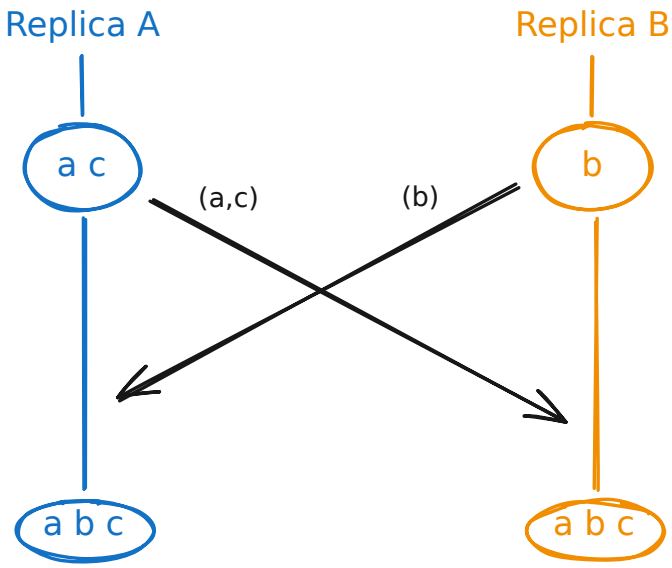

Figure 2.2: Example of a CRDT set merging operation.

When two replicas R_i and R_j exchange states, their state vectors are merged element-wise using the max function:

$$C_i[k] = \max(C_i[k], C_j[k]) \quad \forall k \in \{1, 2, \ldots, n\}$$

The total counter value at R_i is the sum of all elements in its state vector:

$$C_{\text{total}} = \sum_{k=1}^{n} c_{ik}$$

To illustrate the state based counter let us make an example. Assume to have three replicas (R_1, R_2, R_3):
Their initial states are:

$$C_1 = [0, 0, 0], \quad C_2 = [0, 0, 0], \quad C_3 = [0, 0, 0]$$

The following updates are applied to the replicas:

1. R_1 increments by 1: $C_1 = [1, 0, 0]$
2. R_2 increments by 2: $C_2 = [0, 2, 0]$
3. R_3 increments by 1: $C_3 = [0, 0, 1]$

The replicas then propagate their states and merge them:

$$C_1 = [1,0,0] + [0,2,0] + [0,0,1] = [1+0+0, 0+2+0, 0+0+1] = [1,2,1]$$

$$C_{\text{total}} = C_1 = C_2 = C_3 = 1+2+1 = 4$$

It can be seen from these two examples that the final state of the replicas is the same. Notice that the difference in the two categories of CRDTs mainly depends on the implementation.

2.3.2 Operation-Based CRDT Counters (CmRDT)

In an operation-based counter ocal operations, such as `increment` or `decrement`, are broadcast to other replicas. Furthermore, the order in which operations are applied is not important, as long as the operations are commutative.

Let us assume to have a set of replicase $R = \{R_1, R_2, \ldots, R_n\}$, and O represent the set of all operations (`increment` or `decrement`).

For any replica R_i:

- Each operation $o_k \in O$ is of the form:

  $$o_k = \text{INC}(k) \quad \text{or} \quad o_k = \text{DEC}(k)$$

  where $k \in \mathbb{Z}^+$ is the magnitude of the operation.

- The state of the counter at R_i is:

  $$C_i = \sum_{o_k \in O_i} o_k$$

  where O_i is the set of operations applied at R_i.

- Operations are propagated to other replicas:

  $$\forall o_k \in O_i, \quad o_k \text{ is broadcast to } R_j \quad (j \neq i)$$

Then, to make an example let us consider replica R_1 applies an increment operation INC(1) and replica R_2 applies an increment operation INC(2).

1. Replica R_1: Applies INC(1), so $C_1 = 1$.
2. Replica R_2: Applies INC(2), so $C_2 = 2$.

Subsequently, each operation is broadcast to the other replica and applied:

3. Locally at replica R_1 after applying the operation from R_2:

$$C_1 = \sum_{o_k \in O_i} o_k = \text{INC}(1) + \text{INC}(2) = 1 + 2 = 3$$

4. The same happens at replica R_2:

$$C_2 = \sum_{o_k \in O_i} o_k = \text{INC}(2) + \text{INC}(1) = 2 + 1 = 3$$

Then we can observe that despite the order of the operations, the final state in each replica is the same, thus ensuring eventual consistency.

2.3.3 CRDTs Metadata

Notice the condition that in the previous examples the operations are commutative. This is not always the case, for example in the case of a set CRDT, where the order of addition and deletion in the set matters. Several concurrency semantics exist, such as Last-Write-Wins Set (LWW Set), which prioritizes the last write in the set.

In case of concurrency metadata such as timestamps or unique identifiers are used to resolve conflicts. Metadata is necessary to identify causality with the concepts of the happens-before relationship and total ordering among updates[3]. The use of metadata can lead to increased memory usage which is evident in unoptimized implementations, and even in modern implementations the growth of metadata is linear with the number of replicas[4][5].

To summarize CRDTs, they are a powerful tool to ensure eventual consistency and high availability. They do not require synchronization, for example in the form of locks or consensus mechanisms. Operations-based CRDTs are more efficient in terms of communication compared to state-based CRDTs, as they only require the delta of the state to be transmitted, but require the use of metadata to resolve conflicts.

The main disadvantages are the high memory usage for CRDTs that store metadata which is not to be disregarded especially in limited memory environments, even though some solutions to this problem have been proposed [4].

Recent advancements, such as the PS-CRDT model [5], optimize communication by integrating publish/subscribe mechanisms. This reduces communication overhead and enhances scalability, making CRDTs more suitable for dynamic environments such as mobile edge computing.

2.4 Distributed Hash Tables

A Distributed Hash Table (DHT) is a DS which implements the functionality of a hash table, as in key-value pairs are stored in the DHT. The responsibility for storing and retrieving the values associated with the keys is distributed among the nodes in the network. This makes DHTs inherently scalable and resistant to failures.

DHTs rely on overlay networks and hashing techniques such as keyspace partitioning, whose most common implementation is consistent hashing, to organize nodes and data. These characteristics make DHTs effective in environments where nodes frequently join or leave the network, a phenomenon known as churn, which is common in mobile and edge computing environments. The two key aspects of their operation are the mapping of keys to nodes and the routing of requests to the appropriate nodes which will be discussed in the following sections.

2.4.1 Consistent Hashing

For example, assume to use a 160-bit hash space, which translates to 40-digit hexadecimal identifiers, and to have an amount of data that cannot be contained within a single node. The solution would be to divide the data into partitions and assign each part to a different node. This problem is solved by consistent hashing [6]: when data is added to the DHT, a hash function such as SHA-1 is applied to the data, obtaining an hash value.

Nodes in the DS are also given an hash value obtained for example from the ip address of the node. Then, each node becomes responsible for a portion of the hash space - which is organized in logical ring of successive hashes - and data is assigned to the node whose hash value is the closest to the hash value of the data.

It also has to be considered that nodes in the cluster can fail, leave or join the DHT. This would mean that the previous division of the hash space is no longer valid, since the space is divided equally among nodes. In a traditional system this would mean redistributing all the data across the nodes, which is a very expensive operation especially from the point of view of the network. With Consistent Hashing, instead, only a portion of the data is redistributed allowing for less data to be moved, thus a more efficient system.

2.4.2 Overlay Network

Despite being a concept independent from DHTs, overlay networks are fundamental for the operation of DHTs, often coming toghether in the same package as in the case of Tapestry [7]. All the nodes in the DHT form the overlay network which

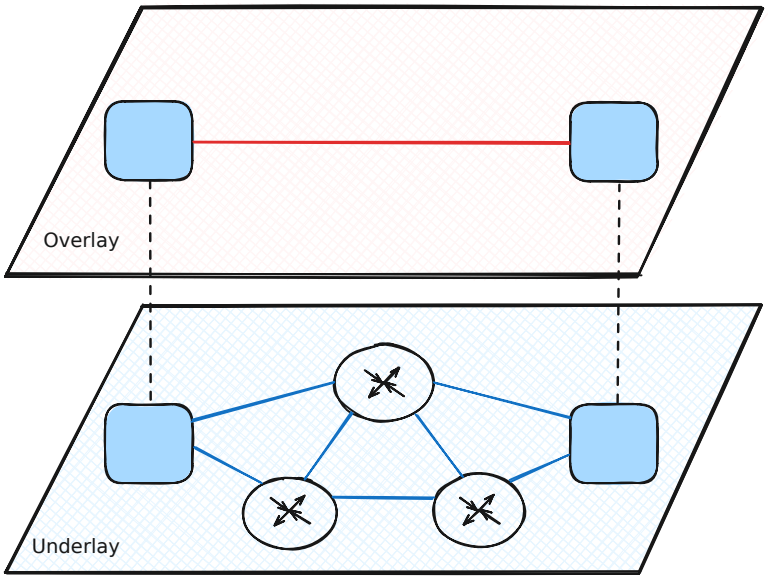

Figure 2.3: Example of overlay network

is a logical network that is built on top of the physical network, providing logical addresses instead of physical ones. Nodes are assigned an unique identifier within the network, which might be derived from the physical ip address of the node but is independent from it. This has the advantage of abstracting the physical network and allowing for more flexibility in the routing of messages and that working with private or public ip addresses is not an issue.

Each node in the overlay network maintains a routing table containing information about other nodes. The routing table is designed such that for any key each node has either a node responsible for that specific key or a node that is closer to the key. In order to reach a key this process typycally requires a $O(\log n)$ number of hops relative to the number of nodes.

2.4.3 DHT Routing

Let us make an example of routing, considering a particular implementation of DHT called Tapestry. Tapestry is a p2p overlay routing infrastructure that provides efficient, scalable and location-independent routing [7].

For the sake of simplicity let us consider an identifier space of hexadecimal strings of length $m = 4$, providing a total of $16^4 = 65{,}536$ possible identifiers. Let the network contain the following four nodes with identifiers $A1F0$, $A2C3$, $B4D5$, and $C3E7$. A data item with identifier $A2F5$ is stored at the node whose identifier is closest to $A2F5$ numerically.

To route a query for $A2F5$, suppose the query originates at node $A1F0$. The process involves matching progressively longer prefixes of the target identifier with entries in the routing table. At each step, the query is forwarded to the node with the closest match for the next unmatched digit in the target identifier.

Starting at node $A1F0$, the query identifies that the first matching prefix is A, and the routing table at $A1F0$ directs the query to node $A2C3$, which shares the prefix $A2$ with the target. At $A2C3$, the next prefix $A2F$ matches partially, and the query is forwarded to the next closest node, which is responsible for $A2F5$. The query terminates once the target node or data item is found.

2.4.4 DHT Performance

DHT are demonstrated to scale well with the number of nodes in the network, as the routing table size grows logarithmically with the number of nodes. With respect to hardware requirements to run a DHT, it has been shown that DHTs can run on commodity hardware, and in particular the computing and power load is low enough to be run for several hours on Nokia n95 mobile phones [8].

2.5 Distributed Databases

We then begin our analysis by looking at distributed databases (DDBs) which specialize in data storage and retrieval across a network of nodes.

DDBs represent a critical evolution in data management technologies, addressing the limitations of traditional centralized database systems by distributing data and computational processes across multiple networked nodes [9]. Unlike monolithic database architectures, DDBs aim to provide horizontal scalability, high availability, and robust fault tolerance which are all properties that were discussed earlier with respect to general distributed systems. Furthermore, databases are optimized for a very high number of transactions and scale well with a high number of nodes.

2.5.1 Apache Cassandra

Apache Cassandra is a NoSQL distributed database developed by Facebook in order to solve the problem of handling high-volume data guaranteeing at the same time resiliency of the system in case of failures and flexibility.

NoSQL databases are designed to be flexible, especially with respect to the disparity of data types and schema definition.

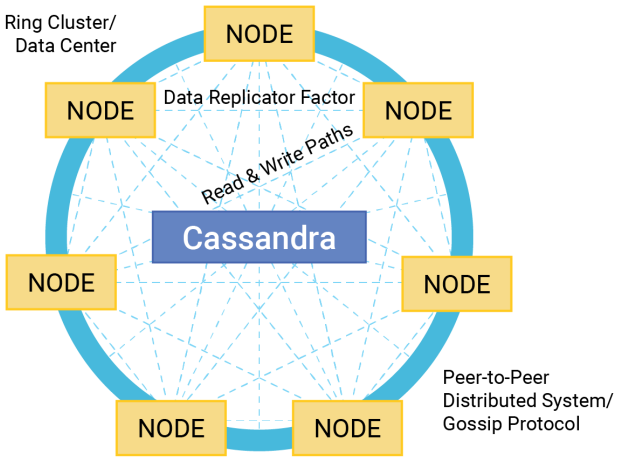

Figure 2.4: Example of Apache Cassandra cluster

2.5.2 Cassandra Architecture

The distributed system of Cassandra is based on a masterless architecture in which all nodes - organized into a a cluster or "ring"- can provide the same functions. For example, in case a coordinator node - responsible for assigning the ownership of data among nodes - fails, another node can be elected in its place. The communication among nodes happens through a peer-to-peer protocol called Gossip protocol, which allows nodes to exchange information about the state of the cluster and detect failures. To summarize, the main components of Cassandra are:

- **Node**: The fundamental unit of Cassandra's distributed system, representing an individual server in the cluster.
- **Cluster**: A collection of nodes that work together to store and manage distributed data.
- **Gossip Protocol**: A peer-to-peer communication mechanism that allows nodes to exchange state information and detect node failures [10].
- **Consistent Hashing**: A technique used for distributing data across nodes, ensuring minimal data redistribution when the cluster configuration changes [6].

2.5.3 Data Replication

In order to guarantee resiliency, data is not stored on a single node but is replicated across multiple nodes. The number of nodes the data is replicated to is defined

by the so called replication factor(RF), which is a parameter that can be set by the user. For instance a replication factor RF of 3 means that the data is replicated across 3 nodes. In case of a node failure then, the requested data can still be retrieved from the other nodes. Replication also offers another advantage: geographical distribution. Data can be replicated across nodes in different parts of the world thus allowing to serve users across different regions with low latency. Data replication works in conjunction with consistent hashing such that multiple nodes are responsible for a portion of the hashing space.

To confirm that the CAP theorem is a continuum[1], Cassandra has a tunable consistency level. One example of consistency level is the quorum level, which requires a majority of nodes to agree on a read or write operation. The consistency level depends on the replication factor:

$$\textbf{QUORUM} = \lfloor \frac{RF}{2} + 1 \rfloor$$

For every request the receiving node becomes the coordinator for that request and is responsible for contacting the other nodes containing the data. The coordinator then waits for the responses from the nodes and if the number of responses reaches the quorum level, the operation is considered successful.

Compared to the CRDT system presented previously, the quorum mechanism is still a form of eventual consistency since it makes sure that data served is the one in the majority of nodes.

2.5.4 Cassandra Hardware Requirements

The following table compares the minimum and recommended hardware requirements for running Apache Cassandra effectively. While lighter requirements may exist, they are generally not suitable for production environments and should only be considered for testing or development purposes.

Table 2.1: Comparison of Minimum and Recommended Hardware Requirements for Apache Cassandra

Hardware Component	Minimum Req.	Recommended Req.
CPU	2 cores	8+ cores
Memory (RAM)	8 GB	32+ GB
Disk	HDD, 1 TB	SSDs with high IOPS

For lighter usage scenarios, such as local testing or small-scale development, Apache Cassandra can run on systems with dual-core CPUs and 8 GB of RAM[11].

Such configurations are not recommended for production, as they can lead to significant performance degradation and instability.

2.6 Blockchain

Blockchain is one of the most significant technologies of the last years as it became especially popular because of its use in cryptocurrencies, with the most famous being Bitcoin.

At its core, it is a decentralized digital ledger that records transactions across a network of computers. Each record in this ledger is called a block, and each block is cryptographically linked to the previous one, forming a chain of information that is extremely difficult to alter or tamper with.

The main characteristics of blockchain are its decentralized nature, allowing for a distributed form of consensus of information without the need for a central authority, and its immutability which means that once a block is added to the chain it cannot be altered. This allows for trust with respect to the information stored in the blockchain, which is publicly available and can be checked by nodes in the system. Furthermore, the consensus mechanism are byzantine fault tolerant, meaning that the system can tolerate a certain number of malicious nodes, making blockchain a natively secure technology.

Even though its initial use cases were limited to cryptocurrencies, as it solved the problems of single point of failure in centralized systems, opaque and potentially manipulated transactions making currency distributed, transparent and - at least in principle - democratic, blockchain technology has been experimented with in a variety of fields such as supply chain management, voting systems and more interestingly in robotics and Internet of Things (IoT) [12].

2.6.1 Blockchain Types

Blockchain can be classified based on two main characteristics: the ability to access the network and the ability to write and commit data.

In the first case the distinction is between public and private blockchains. In public blockchains, anyone can access the network and participate in the consensus mechanism. Private blockchains, on the other hand, allow only selected nodes to participate in the network.

In the second case the distinction is between permissioned and permissionless blockchains. Permissionless blockchains allow anyone to write and commit data to the blockchain by default. In permissioned blockchains, usually used in single

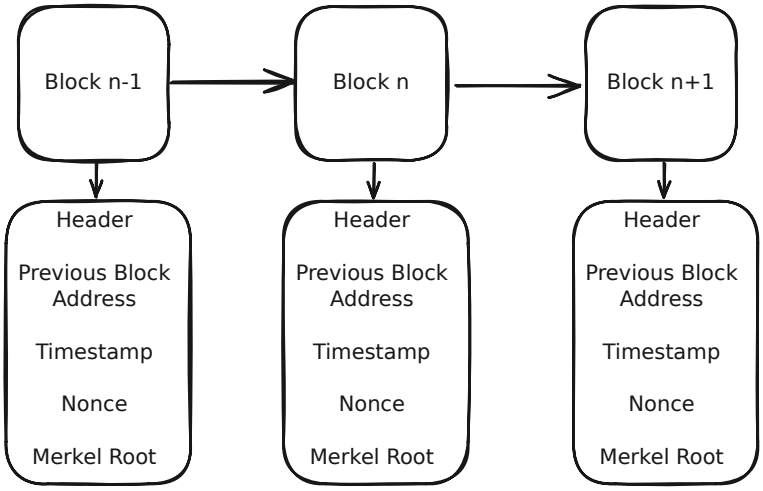

Figure 2.5: Representation of a blockchain

administrative contexts such as enterprises, only few nodes are allowed to write and commit data.

The combination of these characteristics leads to four types of blockchains[12]:

- **Public Permissionless**: allows anyone to participate, write and commit. Examples include Bitcoin and Ethereum.
- **Public Permissioned**: anyone can enter and read transactions, but only selected nodes can write and commit. A possible use case is a voting system.
- **Private Permissionless**: not very common, possible use cases are internal enterprise collaborations where network access is restricted but participants can freely submit transactions.
- **Private Permissioned**: closed entry only few selected nodes can write. Usually used in enterprise environments, an example is Hyperledger.

2.6.2 Blockchain elements

The blockchain is based on three main elements: blocks, hashes and miners. The block is the container of the data on the distributed ledger, it can be logically equiparated to a data structure containing the list of transactions. The blocks are linked to each other through a cryptographic hash. Every block has an associated block header containing various information regarding the previous and current block.

Let us take Bitcoin as an example: the contents of the header are the hash of the previous block, the hash of the current block, the timestamp of the block, the difficulty level, the merkle root of the transactions in the block and finally the nonce. The hash of the previous block is a 256-bit number obtained by applying the SHA-256 algorithm to the transactions of the previous block plus its header. In this way it can be assured that the block is linked to the previous one and that the chain is not tampered with, for example by inserting fake blocks containing fraudulent transactions.

The merkle root is a hash of all the transactions in the block, the difficulty level is a number that determines how difficult it is to find a valid hash for the block and will be explained in more detail in the consensus algorithms section. The nonce is a random number that is used to find a valid hash for the block.

Finally, the miners are the nodes in the network that are responsible for adding new blocks to the blockchain. They do so by solving a complex mathematical problem that usually requires a lot of computational power. In the next section we will discusse how these elements interact with each other and how consensus on the distributed ledger is reached.

2.6.3 Blockchain Consensus Algorithms

In this section we will discuss the answers to the following questions: how are blocks validated and added to the blockchain? How is consensus reached and maintained in the blockchain network? What is the defense against malicious nodes in the network?

In order for a block to be added to the blockchain, it must be validated by the network. This translates to finding a valid hash for the block. For a hash to be considered valid, it must be lower than a target value called difficulty level, a 256-bit number which is dynamically adjusted in order to guarantee a transaction time of around 10 minutes.

The process of finding the valid hash is called mining. The miners in the network compete to find the valid hash using their computational power. In order to find the valid hash, which is computed by hashing both the contents of the block and of its header, the only value that can be changed is the nonce.

The process of mining is shown in the following pseudo code:

Algorithm 1 Finding a Valid Hash in Bitcoin Proof of Work

Input: Block header data B, Target T
Output: Valid nonce N and hash H
Initialize $N \leftarrow 0$ ▷ Start with nonce 0
while true **do**
    $H \leftarrow$ SHA-256(SHA-256($B + N$)) ▷ Double SHA-256 of the block header and nonce
    if $H \leq T$ **then**
        Return N, H ▷ Valid hash found
    end if
    Increment $N \leftarrow N + 1$ ▷ Try the next nonce
end while

In simple terms, set the nonce to a value, obtain the hash, check if the hash is lower than the target value, if not increment the nonce and try again. This process is very computationally intensive since no optimized algorithm for finding the hash exists: only the brute force approach is possible.

After a valid hash is found, the block is broadcasted to the network and the other nodes validate the block by checking the hash and the transactions contained in the block. If the block is valid, it is added to the blockchain and the process starts again for the next block. The miner that found the solution is rewarded with a certain amount of cryptocurrency for the computational (and power consuming) effort.

This consensus mechanism is called Proof of Work (PoW).

2.6.4 Proof of Work

In proof of work the guarantee of validity of the block is given by the high computational power required to find the valid hash. Many nodes are incentivized to participate in the network because of monetary reward.

This in turn increases the security of the network as the more nodes participate the more computational power is available in the network. In order to tamper with the blockchain a malicious node would need to have more computational power than the rest of the network combined. Even if a malicious node were to find a valid hash once, it would have to find it again for the next block and so on, which is highly unlikely.

Furthermore, in the case of a fork in the blockchain, so two parallel chains are created, the valid chain would be the one with the most computational power behind it, thus validanting blocks more rapidly. After some time the longest chain would be the legit one, and the network would converge to it as only the longest chain is considered valid.

This mechanism, even though secure, has some disadvantages: it is very computationally intensive and power consuming, thus making it intrinsically not scalable.

In order to overcome these downsides, other consensus mechanisms have been developed, such as Proof of Stake, which will be discussed in the next section.

2.6.5 Proof of Stake

Proof of Stake (PoS) is an alternative consensus mechanism to PoW that aims to address the energy consumption and scalability issues of PoW. In PoS, validators are chosen based on the number of tokens they hold and "stake" in the network. The more tokens a validator holds, the higher the chance of being selected to validate the next block.

Unlike PoW, there is no computational puzzle to solve - validators simply verify transactions and create new blocks when selected. The security comes from validators having their stake at risk, rather than from expended computational power. This approach is more energy-efficient than PoW, as it does not require the same level of computational power. However, PoS has been criticized for potentially leading to increased centralization, as wealthy stakeholders It is evident how the tradeoff between scalability and decentralization is a fundamental aspect of blockchain technology.

2.6.6 Practical Byzantine Fault Tolerance

Other consensus mechanisms have been proposed with a particular aspect to solve with respect to PoW and PoS. One such algorithm is Practical Byzantine Fault Tolerance (PBFT). PBFT is a consensus mechanism designed to solve the Byzantine Generals Problem in distributed systems.

The Byzantine Generals Problem refers to the challenge of achieving coordinated agreement in a system where some components may act maliciously or fail arbitrarily. It illustrates the difficulty that a set of generals faces when trying to form a unified strategy while communicating solely via messengers who might be unreliable or deceptive. A solution must ensure that loyal generals can consistently agree on a single plan, despite any subset of traitorous or failing generals interfering with communications. This concept lies at the heart of many modern consensus mechanisms, as it highlights the critical need to tolerate arbitrary faults and potential malice within distributed networks.

Unlike PoW or PoS, PBFT achieves consensus through a voting process among known validators. PBFT can tolerate up to f faulty nodes in a network of 3f + 1 total nodes, providing Byzantine fault tolerance. While it offers high transaction

throughput and immediate finality, it requires significant communication overhead between nodes and is better suited for smaller networks with known participants.

The communication complexity of PBFT grows quadratically with the number of nodes ($O(n^2)$), making it impractical for large networks with hundreds or thousands of nodes. This quadratic scaling is due to the requirement that each node must communicate directly with every other node during consensus.

2.6.7 Blockchain Observations

From the analysis of blockchain technology several observations can be made. First, blockchain utilizes consensus mechanisms to provide data consistency across nodes, making it particularly suitable when immutability and transparency are needed.

The immutable nature of the ledger and the built-in cryptographic protection make blockchain inherently secure against tampering and malicious attacks. Furthermore, the ability to operate without a central authority makes it resilient to single points of failure.

However, blockchain faces significant issues in terms of scalability and resource consumption. The computational requirements for consensus mechanisms like PoW are substantial, making them impractical for resource-constrained environments such as mobile or IoT devices. Even more efficient consensus mechanisms like PoS and PBFT still require significant computational resources compared to other distributed system approaches.

The communication overhead in blockchain networks can also be substantial. For instance, PBFT's quadratic communication complexity makes it unsuitable for large networks, while PoW's requirement to broadcast blocks to all nodes can lead to significant network traffic. These factors make blockchain less suitable for applications requiring high throughput or real-time performance in resource-limited environments.

2.7 Multicast

Despite not being a technology used in the context of geographically distributed systems, this section will present the concept of multicast communication, which is relevant for the efficient communication in a private network. Because of policies on network efficiency and security, multicast traffic is blocked by default in the public internet, since it is a limited form of broadcast that can quickly saturate network links and overwhelm devices, as well as posing security concerns.

In traditional unicast communication, a sender transmits data packets to a single

receiver. While this approach works well for one-to-one communication, it becomes inefficient when the same data needs to be sent to multiple receivers. In a scenario where a video stream needs to be delivered to thousands of users unicast would require the sender to create and transmit thousands of separate data streams. This would lead to significant bandwidth consumption and network congestion, easily saturating network links.

Multicast addresses this issue by allowing a single data stream to be sent to multiple receivers simultaneously. It operates at the network layer (Layer 3) of the OSI model and utilizes a special class of IP addresses known as multicast addresses. These addresses range from 224.0.0.0 to 239.255.255.255 in IPv4. Multicast enables efficient data distribution by allowing routers to replicate and forward packets only to the network segments where there are interested receivers, thus conserving bandwidth.

2.7.1 Multicast Routing

To illustrate how multicast works, consider an example where a node subscribes to a multicast channel and receives a packet. The process begins with the receiver node expressing its interest in a specific multicast group by sending an Internet Group Management Protocol (IGMP) join message to its local router. The router, upon receiving the IGMP join message, updates its multicast forwarding table to include the new receiver in the multicast group.

The local router then communicates with upstream routers using a multicast routing protocol like Protocol Independent Multicast (PIM) to ensure that multicast traffic for the requested group is forwarded to it. The upstream routers, in turn, update their forwarding tables and ensure that the multicast traffic is propagated down the correct branches of the distribution tree.

When the sender transmits a multicast packet, it is forwarded by the routers along the distribution tree. Each router checks its multicast forwarding table to determine the outgoing interfaces that have subscribed receivers. The packet is then replicated and forwarded only on those interfaces, ensuring that it reaches all interested receivers without unnecessary duplication.

As the packet traverses the network, each router performs similar checks and forwarding actions until the packet reaches the final receivers. This efficient packet replication and forwarding mechanism significantly reduces the bandwidth consumption and processing overhead compared to unicast, especially in scenarios with a large number of receivers.

In summary, multicast provides an efficient solution for one-to-many communication. It is particularly useful and light on resources for distribution of data to

multiple receivers in a private network, such as in a local area network.

2.8 Comparison

According to the problem at hand, the solution for discovery should be highly available and fault tolerant, it should be completely decentralized to avoid single points of failure and should be lighweight with respect to both hardware and network resources. A possible use case is indeed a Unmanned Ground Vehicle (UGV) that needs to offload some computation to the edge, so the discovery mechanism should be able to work in a resource constrained environment.

We assume a dynamic network where nodes can join and leave the network at any time and are expected to do so, so behaviour in presence of churn should be considered. Furthermore, the energy efficiency of the solution is a key aspect to consider since different technologies have different power requirements.

With these requirements in mind it is now possible to summarize the characteristics of the technologies and compare them.

Table 2.2: Comparison of Technologies for Discovery in Distributed Systems

Property	**CRDTs**	**DHT**	**Distributed DBs**	**Blockchain**
Key Features	SEC and conflict resolution without synchronization	Consistent hashing, churn resistant, overlay network	High throughput	Distributed ledger with immutable, transparent and secure transactions
Scalability	Hundreds of nodes[5]	Thousands of nodes[7], O(log n) for routing table size	Tens of Thousands of nodes[13], up to 450 per cluster	Limited scalability due to the overhead of consensus
CPU	Low, comparable to DHT	Low, can be run on nokia n95 mobile phones[8]	2 minimum, 8+ recommended[11]	High, even though GPU or ASIC computation is more popular
RAM	High in unoptimized version, 110MB for 1000 nodes in some conditions[4]	Less than 128MB for 1000 nodes[8]	8GB minimum, 32+ GB recommended[11]	GBs order of magnitude

Table 2.2 presents the key features, hardware requirements and power consumption. As discussed in the previous sections all presented technologies are highly available and partition tolerant so the focus of the analysis will be the hardware requirements and power consumption.

CRDTs used as data structures guarantee eventual consistency without the need for synchronization. Appropriate kind of CRDTs to share a set of address information such as a LWW Set. need to also share metadata, whose unbounded storage can be an issue from the point of view of memory usage and cannot be ignored. Enes et al. [4] consider this problem in a environment under churn.

By implementing a garbage collection mechanism, the authors show that the memory usage in a 4 node network sharing a add-wins set - which is functionally equivalent to a LWW Set but gives priority to the addition of elements and uses unique tags instead of timestamps - after 10000 add operations is less than 2MB, while the unoptimized version reaches this amount around 3000 add operations. It is shown that memory usage also increases with the number of nodes due to the increased metadata received from other nodes. Nevertheless, the authors show that when stabilizing the system and increasing the number of nodes from 1 to 8, the memory usage increases by around 110KB. Projecting this result to 1000 nodes sets the memory usage to around 110MB, which is a reasonable amount of memory for a modern device. Further research is needed to understand power and CPU requirements.

DHT stores key-value pairs across the network, where each node is responsible for a portion of the keyspace. Consistent hashing ensures minimal data redistribution in case of churn. DHT implementation usually come togheter with an overlay network, such as Tapestry which provides efficient, scalable and location-independent routing. Usual implementations are scalable for a high number of nodes, as lookup time and data storage grow logarithmically. Zhao et al. [7] show that Tapestry performs well on the PlanetLab network, a global-scale testbed for distributed systems research. Even in conditions of churn, Tapestry is able to maintain a high level of performance with 1000 nodes communicating in the network. With respect to hardware and power requirements, Ou et al. [8] demonstrate that the Kademlia DHT, which differs from Tapestry mainly in the routing algorithm, is able to be run by nokia n95 mobile phones from 2009 for several hours, showing that DHTs are suitable for resource constrained environments.

Distributed DBs are designed to provide horizontal scalability, high availability, and robust fault tolerance. Internally they use concepts such as consistent hashing in Apache Cassandra or CRDTs in Riak[14]. They provide high transaction throughput but are resource intensive, requiring for example a minimum of 8GB and 2 cores to run Cassandra effectively [11]. They are designed to handle a

high number of requests and despite being decentralized, they are expected to be supervised by a single administrative entity, making them less suitable for a completely decentralized environment.

Blockchain technology guarantees immutability, integrity and security of the data stored in the ledger. The main issue with blockchain is the high computational power required for consensus mechanisms like PoW, which require specialized hardware such as GPUs or ASICs in order to solve in less time the cryptographic nonce puzzle. Evolutions like PoS decrease power consumption and thus hardware requirements but centralize validators in the network.

Research in the field of robotics has been conducted in order to benefit from the security aspects of blockchain in industrial, swarm, UAV and multirobot systems, as discussed by Ferrer et al. [12]. They conclude that for use cases of mobile teleoperated robots limited energy, storage and computation capacity are major impediments to blockchain integration due to computationally intensive consensus mechanisms.

Strobel et al. [15] integrate DLT technology with swarm robotics in the use case of a search and rescue mission. They show that the DLT implementation is able to perform the task even in presence of byzantine agents, but conclude that it will not have a positive influence on energy consumption.

Sedlmeir et al. [16] show the amount of energy consumed by PoW blockchains, with Bitcoin estimated to require power in the order of Giga Watts worlwide, and at least three orders of magnitude more energy per transaction than PoS blockchains. It is also shown that some blockchains such as Hedera[17] are very low in energy consumption, but make it so by being permissioned networks and thus centralized.

In conclusion, all the technologies presented are highly available and partition tolerant. Blockchain is the most secure technology but also the most resource intensive one, requiring high computational power and energy consumption. Both conditions are not suitable for our use case. The immutability of transactions is not relevant to the problem of address discovery since only the current address is relevant. A simpler solution, should it be needed, would be to log locally the history of discovered nodes.

Distributed DBs are designed for high transaction throughput and are expected to be supervised by a single administrative entity, making them not suitable for a completely decentralized environment. Furthermore they have considerable hardware requirements (minimum of 2 CPU cores, 8GB RAM[11]) which makes them not suitable for resource constrained environments.

DHTs distribute the responsibility for storing and retrieving data in the network.

They are designed with churn scenarios in mind and are able to run on commodity hardware such as mobile phones with reasonable memory and energy consumption. Furthermore, they often come with an overlay network such as Tapestry that provides efficient routing and allows for operation in both private and public IP networks.

CRDTs provide eventual consistency without the need for synchronization. Their memory usage is on par with the one of DHTs in case of optimized implementations[4]. CRDTs are ideal in case of concurrent operations since they resolve conflicts without central coordination, a characteristic that is not needed in the use case of address discovery since the address of nodes should be propagated by nodes themselves. Indeed if a node is not able to communicate its address to the network, the network would not be able to communicate with it and thus offload computations to it.

After comparing the technologies we can conclude that DHTs are currently the most suitable technology for the problem of address discovery in a distributed system. They address the scalabilty, low resource and power requirements, with mature overlay network implementations such as Tapestry they provide network independent routing and are able to handle churn scenarios.

Chapter 3

Kubernetes

As the computational demand for modern applications continued to grow in recent years, a shift in paradigm occurred in the way software was developed, deployed, and managed. Traditional monolithic applications, incorporating all the business logic and functionality into a single codebase and deployed on physical servers could not handle the need for scalability, reliability, and agility that modern applications demanded.

This led to the rise of microservices architecture, where applications are broken down into smaller, loosely coupled services that can be developed, deployed, and scaled independently. Vertical scaling, where a single server is upgraded with more resources to handle increased load, was no longer sufficient. Instead, horizontal scaling became the preferred approach, where multiple instances of a service are deployed across a distributed infrastructure via virtualization or containerization technologies. Managing all the instances of these services across a distributed infrastructure is a complex task that cannot be done manually: imagine handling thousands of containers that might fail and need a restart. This is where container orchestration systems come into play.

Kubernetes (often abbreviated as K8s) emerged from Google's internal container orchestration system called Borg, which Google had been using for over a decade to manage its vast infrastructure [18]. In 2014, Google open-sourced Kubernetes, making it available to the broader community. Since then, it has grown to become the de facto standard for container orchestration and is now maintained by the Cloud Native Computing Foundation (CNCF).

The development of Kubernetes addressed a critical need in the industry. As more organizations began adopting containerization technologies like Docker, they faced significant challenges in managing and orchestrating large numbers of containers across distributed environments. Before Kubernetes, deploying and managing

containerized applications at scale was a complex and labor-intensive process that often resulted in operational inefficiencies and reliability issues [19].

3.1 The Cattle Model and Scalability

Traditional infrastructure management followed what is now referred to as the "pets model", where servers were individually named, carefully maintained, and treated as unique, indispensable entities. When issues arose, administrators would diagnose and repair these servers, taking great care to preserve their state and configuration, much like caring for a pet [20].

Kubernetes embodies a fundamentally different paradigm known as the "cattle model". In this approach, computing resources are standardized, stateless, and disposable - treated as undifferentiated members of a herd rather than unique individuals. This philosophical shift represents one of Kubernetes' core design principles and enables many of its most powerful features.

Under the cattle model, individual nodes and containers are considered expendable. When a container or node fails or exhibits problematic behavior, rather than attempting to diagnose and repair it, the system simply terminates and replaces it with a fresh instance. This approach offers several significant advantages for modern distributed systems:

3.1.1 Immutability and Declarative Configuration

The cattle model embraces immutability as a fundamental principle. Containers are treated as immutable units that should not be modified after deployment. Instead of updating running containers, Kubernetes creates new ones with the updated configuration and gradually replaces the old ones. This approach ensures consistency and predictability across environments and enables powerful features like rollouts and rollbacks [21].

Configuration is handled declaratively rather than imperatively. Administrators specify the desired state of the system - how many replicas should run, what resources they should have access to, how they should be networked - and Kubernetes continuously works to ensure the actual state matches this desired state.

3.1.2 Self-healing Capabilities

Perhaps the most significant benefit of the cattle model is Kubernetes' ability to self-heal. When a node fails, becomes unresponsive, or cannot maintain expected performance, Kubernetes automatically reschedules the affected workloads onto healthy nodes. Similarly, when individual containers crash or fail health checks, Kubernetes automatically restarts them or creates replacement instances [22].

This self-healing capability minimizes downtime and reduces operational burden. System administrators no longer need to perform middle-of-the-night interventions to restart failed services or provision new servers when hardware fails. Instead, they can focus on higher-value activities while Kubernetes handles routine recovery tasks automatically.

3.1.3 Horizontal Scaling

The cattle model facilitates horizontal scaling - adding more instances rather than increasing the resources of existing ones. Kubernetes enables services to scale out dynamically in response to increasing demand and scale in when demand decreases. This elasticity improves resource utilization and ensures applications remain responsive under varying loads [23].

Horizontal scaling also improves fault tolerance through redundancy. By running multiple instances of each service across different nodes, Kubernetes ensures that the failure of any single container or node does not bring down the entire service. Traffic is automatically redistributed to the remaining healthy instances while replacements are spun up.

3.2 Kubernetes Architecture

Kubernetes follows a distributed architecture that enables its scalability, resilience, and flexibility. At its core, the system employs a master-worker pattern with a clear separation between the control plane and the data plane. This architectural approach allows Kubernetes to manage thousands of containers across hundreds of nodes while maintaining operational stability.

3.2.1 Control Plane Components

The control plane serves as the brain of the cluster, managing the overall state and making global decisions about cluster operation. It maintains a record of all resources within the cluster, monitors their status, and takes action to ensure the actual state aligns with the user's defined intent. These components are typically deployed in a high-availability configuration across multiple machines to ensure fault tolerance [22].

The control plane consists of several key components:

- **API Server** - The front-end interface for the Kubernetes control plane that exposes the Kubernetes API. All communications, both internal (between components) and external (from users), flow through the API Server. It

validates and processes RESTful requests and updates the corresponding objects in etcd.

- **etcd** - A distributed, consistent key-value store that serves as Kubernetes' primary datastore for all cluster data. Its consistency and high-availability properties ensure that the cluster state is reliably maintained even during hardware or network failures.

- **Scheduler** - Responsible for assigning newly created pods to nodes. The scheduler makes placement decisions based on resource requirements, hardware/software/policy constraints, and other specifications. It embodies sophisticated algorithms to ensure optimal workload distribution across the cluster.

- **Controller Manager** - Runs controller processes that regulate the state of the cluster. These controllers include the Node Controller (monitoring node health), Replication Controller (ensuring the correct number of pods are running), Endpoints Controller (populating endpoint objects), and many others. Each controller implements a control loop that watches the shared state and makes changes to move the current state toward the desired state.

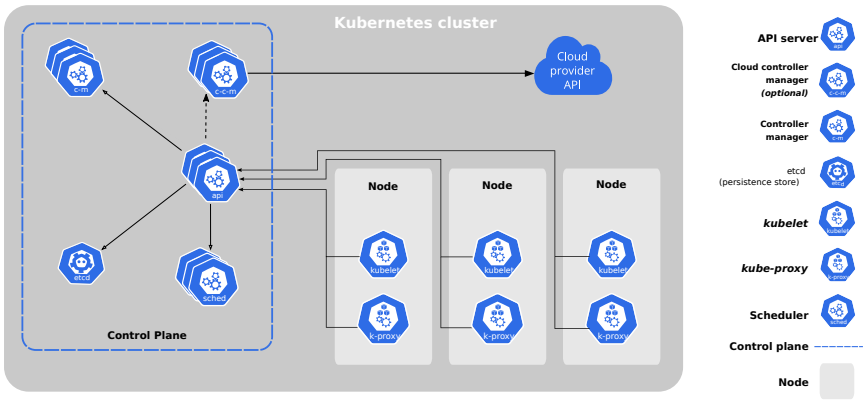

Figure 3.1: Kubernetes architecture

3.2.2 Data Plane Components

The data plane comprises worker nodes that execute the actual workloads. These nodes host the containerized applications and handle networking and storage operations. Worker nodes operate largely autonomously, with the ability to continue

running existing workloads even during temporary disconnections from the control plane [23].

Each worker node in the data plane runs several essential components:

- **Kube-proxy** - A network proxy that runs on each node, implementing part of the Kubernetes Service concept. It maintains network rules that allow network communication to pods from inside and outside of the cluster. Kube-proxy uses the operating system's packet filtering layer or runs in userspace mode to handle forwarding.
- **Kubelet** - The primary node agent that runs on each node. It takes a set of PodSpecs provided by the API server and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers that weren't created by Kubernetes.
- **Container Runtime** - The software responsible for running containers. While Docker was historically the most common runtime, Kubernetes now supports multiple container runtimes through the Container Runtime Interface (CRI). Options include containerd, CRI-O, and others. The container runtime handles low-level container operations such as image pulling and container execution.

3.2.3 API-driven Design

The entire Kubernetes system is built around a unified API. All operations - from deploying applications to querying resource status - are performed through this API. This API-centric approach provides several architectural advantages:

First, it enables a clean separation between the control plane and client tools. Any authorized client can interact with the cluster through the same API, whether it's the official command-line tool (kubectl), a custom script, or a third-party application.

Second, the API provides a consistent way to interact with all resources in the system. The same patterns apply whether managing core resources like pods and services or custom resources defined through extensions.

Third, the API server acts as the sole point of truth within the cluster. All changes to cluster state must go through the API server, which validates them and stores the resulting state in the persistent storage backend (etcd). This centralized state management ensures consistency across the distributed system [24].

3.3 Kubernetes Fundamental Components

Kubernetes organizes computational resources and applications using a set of fundamental abstractions that form the building blocks of any deployment. These

abstractions represent physical resources, application workloads, networking, and configuration.

3.3.1 Pods

The Pod is the smallest and most basic deployable unit in Kubernetes. A Pod represents a single instance of a running process in a cluster and encapsulates one or more containers, shared storage, network resources, and specifications for how to run the containers [22].

Pods are designed to be ephemeral, disposable entities. When a Pod is deleted or fails, Kubernetes does not resurrect it - instead, it creates a new Pod to replace it. This aligns with the cattle model discussed earlier and is fundamental to understanding Kubernetes' approach to application management.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.19
    ports:
    - containerPort: 80
```

Listing 3.1: Example Pod Manifest

This manifest defines a simple Pod running an Nginx container. The Pod specification includes resource requests and limits - a crucial aspect for proper scheduling and resource management within a cluster.

3.3.2 Deployments

While Pods are the basic unit, directly creating Pods is rarely the ideal approach. Instead, Kubernetes offers higher-level abstractions that manage Pods automatically. The Deployment controller provides declarative updates for Pods and ReplicaSets, ensuring that the desired number of Pod replicas are running at all times [23].

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
```

Listing 3.2: Example Deployment Manifest

This manifest creates a Deployment that maintains three replicas of the Nginx Pod. If any Pod fails or is deleted, the Deployment controller automatically creates a new one to maintain the desired state of three replicas.

3.3.3 Services

Kubernetes Pods are ephemeral - they can be created, destroyed, or rescheduled based on cluster conditions. This dynamic nature makes it challenging for applications to reliably communicate with each other. The Service resource solves this problem by providing a stable network endpoint to access a dynamic set of Pods [25].

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
```

```
12   type: ClusterIP
```

Listing 3.3: Example Service Manifest

This manifest creates a Service that provides a stable IP address and DNS name for accessing the Nginx Pods. The Service selects Pods with the label `app: nginx` and forwards traffic to port 80 on those Pods.

3.3.4 DaemonSets

DaemonSets provide a way to ensure that a specific Pod runs on all (or a subset of) nodes in a cluster. As nodes are added to the cluster, Pods are automatically added to them; as nodes are removed, the Pods are garbage collected. This pattern is particularly useful for cluster-wide operations such as log collection, monitoring, and networking plugins [21].

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: node-monitoring
5    labels:
6      app: monitoring-agent
7  spec:
8    selector:
9      matchLabels:
10       app: monitoring-agent
11   template:
12     metadata:
13       labels:
14         app: monitoring-agent
15     spec:
16       containers:
17       - name: monitoring-agent
18         image: monitoring/agent:v1
19         volumeMounts:
20         - name: varlog
21           mountPath: /var/log
22       volumes:
23       - name: varlog
24         hostPath:
25           path: /var/log
```

Listing 3.4: Example DaemonSet Manifest

This manifest creates a DaemonSet that runs a monitoring agent on every node in the cluster. The agent has access to the host's `/var/log` directory to collect logs. DaemonSets are particularly relevant to the FLUIDOS project as they provide a mechanism for deploying components that need to run on every node, such as networking plugins or discovery agents.

3.3.5 Custom Resources and Controllers

Kubernetes provides an extension mechanism through Custom Resource Definitions (CRDs) and custom controllers. This enables the definition of new resource types that extend the Kubernetes API, allowing for domain-specific abstractions that integrate seamlessly with the platform [26].

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: knownClusters.discovery.fluidos.eu
spec:
  group: discovery.fluidos.eu
  names:
    kind: KnownCluster
    plural: knownClusters
    singular: knownCluster
    shortNames:
    - kc
  scope: Namespaced
  versions:
  - name: v1alpha1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              clusterID:
                type: string
              endpoint:
                type: string
              lastSeen:
                type: string
                format: date-time
```

```
32                required: ["clusterID", "endpoint"]
```

Listing 3.5: Example Custom Resource Definition

This CRD defines a new resource type called ‘KnownCluster‘ that represents a discovered FLUIDOS node. The custom controller for this resource would implement the logic for managing the lifecycle of these resources, including discovery, monitoring, and cleanup operations.

In the context of the FLUIDOS project, these custom resources are crucial for representing discovered nodes in the network and enabling cross-cluster communication [27].

3.4 Kubernetes Networking

Kubernetes networking provides the foundation for communication between containers, pods, services, and external clients. Understanding this networking model is essential for grasping how applications communicate within and across clusters, especially in distributed environments like those addressed in the FLUIDOS project.

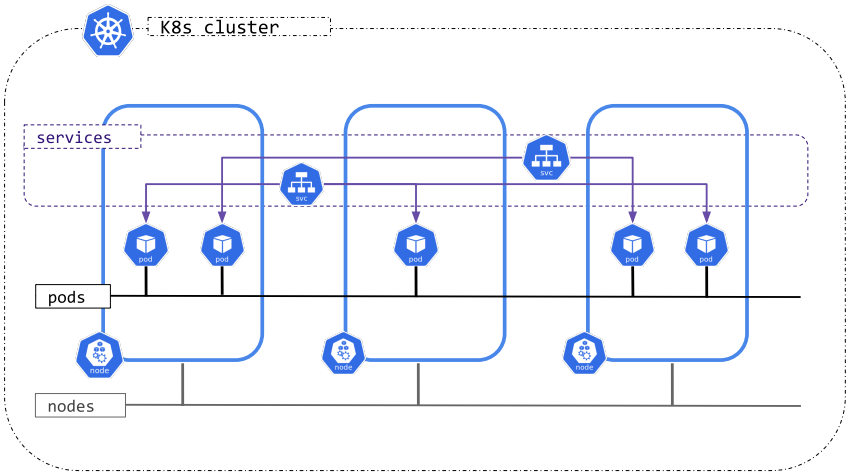

Figure 3.2: Kubernetes networking model

3.4.1 Networking Model Fundamentals

The Kubernetes networking model rests on a set of fundamental principles designed to simplify application communication while maintaining necessary isolation. At its core, the model ensures every pod receives a unique IP address across the

cluster. Containers sharing a pod also share a network namespace, allowing them to communicate via localhost. The model guarantees that pods on the same node can communicate with each other directly without network address translation (NAT), as can pods across different nodes. Perhaps most importantly, the IP address that a pod sees itself as is the same IP that other pods or services see it as.

This flat networking approach creates a uniform communication space that eliminates many traditional networking complexities. Applications no longer need to discover host IPs or manage port mappings, as each pod becomes directly addressable from any other pod in the cluster. This simplicity enables developers to design applications as if they were running on traditional networks while gaining the scalability and reliability benefits of containers.

3.4.2 Container Network Interface

While Kubernetes defines the networking model, it delegates implementation to Container Network Interface (CNI) plugins. This approach follows a clear separation of concerns, allowing Kubernetes to focus on orchestration while specialized networking solutions handle the complex task of network configuration. The CNI specification provides a common interface for third-party networking solutions to integrate with Kubernetes, creating a rich ecosystem of options.

Various CNI implementations have emerged to address different requirements and environmental constraints, for example Calico, Flannel, Cilium [28], [29], [30].

3.4.3 Pod-to-Pod Communication

Pod-to-pod communication forms the foundation of Kubernetes networking. When pods reside on the same node, they communicate through the node's internal bridge network. Each pod connects to this bridge via a virtual Ethernet pair, with one end in the pod's network namespace and the other in the root namespace connected to the bridge. This direct connection allows efficient local communication without leaving the node.

Communication between pods on different nodes is more complex and varies depending on the CNI implementation. Some solutions, like Flannel, create overlay networks using technologies such as VXLAN or GRE tunnels to encapsulate traffic between nodes. Others, like Calico, program direct routes using routing protocols like BGP, avoiding the encapsulation overhead at the cost of more complex network configuration. Regardless of implementation details, the result appears the same to containers: seamless communication across the cluster with pods addressable by their unique IP addresses.

3.4.4 Service Discovery and Load Balancing

The ephemeral nature of pods presents challenges for building reliable applications. Pods can be created, destroyed, or rescheduled at any time, making direct pod-to-pod communication unreliable for long-lived connections. Kubernetes addresses this challenge through Services, an abstraction that provides a stable endpoint for a dynamic set of pods.

Services function as an internal load balancer, selecting a set of pods based on labels and providing a single stable IP address and DNS name to access them. The kube-proxy agent, running on every node, implements this abstraction by creating network rules that distribute traffic to the selected pods. These rules vary based on the operation mode - userspace, iptables, or IPVS - with each offering different performance and feature tradeoffs.

Kubernetes also implements an internal DNS service that automatically registers each service. When a service is created, it receives a DNS entry in the format <service-name>.<namespace>.svc.cluster.local, allowing pods to discover services by name rather than by IP address. This DNS-based discovery mechanism enables service-oriented architectures where components find each other through well-known names, simplifying application configuration and enabling dynamic reconfiguration.

While service discovery works seamlessly within a single cluster, it becomes more challenging in multi-cluster environments. The standard Kubernetes DNS does not span cluster boundaries, creating a significant limitation for distributed applications. The discovery protocols evaluated in this thesis address this gap by extending Kubernetes' service discovery capabilities to work across cluster boundaries, enabling truly distributed applications.

3.4.5 Network Policies and Security

By default, Kubernetes allows unrestricted communication between all pods in a cluster. This open approach simplifies initial deployment but falls short of security best practices, particularly in multi-tenant environments. Network Policies address this concern by providing a mechanism to control traffic flow between pods, functioning similarly to a network firewall.

Network Policies define rules specifying which pods can communicate with each other and on which ports. These policies can filter traffic based on pod labels, namespaces, IP ranges, and port numbers, allowing fine-grained control over network communication. For example, a policy might restrict a database pod to accept connections only from specific application pods and deny all other traffic, implementing the principle of least privilege.

3.4.6 External Communication

Kubernetes provides several mechanisms for pods to communicate with external services and for external clients to access services within the cluster. For outbound communication, pods typically reach external networks directly, with the node performing source NAT on the outgoing traffic. This allows external services to respond to the request without requiring direct routing to the pod network.

For inbound communication, Kubernetes offers several service types with increasing levels of external accessibility. NodePort services expose an application on a static port on each node's IP address, making it accessible from outside the cluster at <node-ip>:<node-port>. LoadBalancer services extend this by provisioning an external load balancer with a public IP in supported cloud environments. For HTTP and HTTPS traffic, Ingress controllers provide more sophisticated routing based on hostnames and paths, often with additional features like SSL termination and authentication.

Chapter 4

FLUIDOS

4.1 Introduction

FLUIDOS represents a paradigm shift in distributed computing, aiming to leverage the vast, underutilized processing capacity at the edge. This capacity is currently scattered across heterogeneous edge devices that struggle to integrate with each other and form a seamless computing continuum [27].

Traditional cloud computing models centralize resources in large data centers, creating a clear separation between edge devices and cloud infrastructure. This model fails to efficiently utilize the growing computational power available at the edge, leading to increased latency for latency-sensitive applications and unnecessary data transfers to distant data centers. Furthermore, the rigid boundary between edge and cloud environments prevents seamless resource utilization across the computing continuum [31].

FLUIDOS overcomes these limitations by enabling the creation of a virtual computing space spanning across multiple physical domains. This allows services started within this virtual space to leverage all available resources within the same virtual domain, regardless of physical location. In the FLUIDOS ecosystem, a service can seamlessly scale based on resource availability across the entire virtual infrastructure, potentially having instances running simultaneously at the telco edge and in cloud datacenters, effectively blurring the rigid cluster boundaries that currently exist [27].

The project builds on the foundation of Kubernetes while extending its capabilities to address the unique challenges of the computing continuum. This chapter explores the FLUIDOS architecture, its key components, and the discovery protocols that are tested in this thesis.

4.2 The FLUIDOS Computing Continuum

While the concept of a computing continuum might appear to exist today with applications already relying on components distributed across different locations—from edge devices to cloud data centers—the FLUIDOS approach to liquid computing introduces three distinctive characteristics that set it apart from existing solutions.

4.2.1 Deployment Transparency

Traditional distributed applications require explicit configuration to deploy each component to a specific target location. In conventional systems, DevOps engineers must predetermine whether a microservice runs at the edge datacenter or in the cloud, with these locations fixed and decided a priori. This rigid deployment model creates significant challenges for dynamic optimization, as any runtime adjustments require complex re-deployment orchestration—a capability that current technologies largely lack.

FLUIDOS revolutionizes this approach through its intent-based interface, which ensures that each microservice starts in the optimal location based on service requirements and infrastructure status. This abstraction layer allows DevOps teams to interact with a single deployment and control point, while the underlying FLUIDOS system automatically places services in the most appropriate locations across the computing continuum. Moreover, the system continuously evaluates placement decisions and can dynamically relocate services as conditions change.

This deployment transparency significantly simplifies operations and enables more efficient resource utilization by automatically matching workload requirements with the most suitable available resources, whether they exist at the edge, in a telco facility, or in the cloud.

4.2.2 Communication Transparency

In traditional distributed systems, communication between microservices varies significantly depending on their relative locations. Services within the same cluster communicate differently from those in separate clusters, requiring distinct networking configurations and security policies. For example, Kubernetes uses ClusterIP services for internal cluster communication but requires NodePort or LoadBalancer services for external communications.

This location-dependent communication model forces developers to explicitly configure services based on their deployment location, further complicating any potential redeployment or scaling operations. While some technologies like message brokers (e.g., Kafka) partially address these issues through publish/subscribe

mechanisms, they cannot guarantee reduced latency and may not be suitable for all application types.

FLUIDOS creates a virtual cluster spanning multiple physical clusters, allowing applications to operate seamlessly across this virtual space. All communications between microservices are mediated by the FLUIDOS virtual network fabric, providing consistent communication primitives regardless of service location. This approach eliminates the need for complex, error-prone configurations that must account for whether services reside in the same or different clusters [27].

With FLUIDOS, two services in the same virtual domain communicate as though they were in the same cluster, even if physically deployed on entirely different infrastructure components. This communication transparency enables more flexible application architectures and simplifies both development and operations.

4.2.3 Resource Availability Transparency

Current container orchestration technologies restrict services to using only the resources available within their own cluster. This limitation applies both during initial deployment and subsequent scaling operations. As a result, services may experience disruptions due to resource constraints in one cluster, even when abundant resources are available in adjacent clusters within the same administrative domain.

This problem, while less significant in cloud datacenters with virtually unlimited resources, becomes critical in edge environments where resources are inherently limited. Edge deployments typically comprise a small number of servers with constrained capabilities, making the ability to leverage nearby resources crucial for maintaining service quality.

FLUIDOS overcomes this limitation by creating a virtual computing space that spans multiple physical domains. Services running in this virtual space can access all resources belonging to the same virtual domain, regardless of physical location. This capability enables seamless scaling based on resource availability across the entire virtual infrastructure.

For example, a service might automatically expand to have instances running simultaneously at the telco edge and in cloud datacenters, effectively erasing traditional cluster boundaries. This resource availability transparency maximizes infrastructure utilization and resilience, particularly important for edge computing scenarios where resources may be limited and heterogeneous.

4.3 Technology Foundation

4.3.1 Kubernetes as the Substrate

FLUIDOS adopts Kubernetes as its technological foundation, a choice driven by several compelling factors. Kubernetes has emerged as the de facto standard for container orchestration, offering a robust platform for managing containerized applications at scale [22]. Its adoption by FLUIDOS leverages several key advantages:

Kubernetes provides a cloud-native approach to application deployment and management, offering unprecedented agility and efficiency compared to traditional virtual machine-based systems. Its scalability allows applications to expand across numerous servers within a cluster, handling substantial traffic loads without downtime. The platform's portability means it functions across various environments, from on-premises installations to multiple cloud providers [23].

Particularly relevant for FLUIDOS is Kubernetes' support for deployments across varying scales. Different Kubernetes distributions accommodate both large-scale deployments in cloud datacenters and small-scale implementations on resource-constrained edge devices, making it an ideal candidate for building the meta-operating system concept that FLUIDOS represents [32].

Additionally, Kubernetes offers flexibility through its extensive feature set for managing containerized applications, including scaling, deployment, updates, and monitoring capabilities, while supporting various container runtimes. Its resource optimization automatically allocates containers based on available resources and workload demands, a critical feature for the resource-constrained environments often found at the edge [21].

The platform's robust community support has created a rich ecosystem of add-ons and tools that FLUIDOS can leverage and extend. However, while Kubernetes forms the technological foundation, the FLUIDOS architecture and its proof-of-concept components are designed with broader applicability in mind, potentially enabling their reuse with alternative technological substrates.

4.4 FLUIDOS Architecture

The FLUIDOS architecture embodies several fundamental characteristics that define its approach to distributed computing:

- **Intent-driven design** allows consumers to assign execution constraints to workloads through high-level policies without requiring knowledge of infrastructural details. This approach extends the "cattle service model" to a broader

scale, treating computational resources as interchangeable units rather than unique entities requiring individual attention.

- **Decentralized architecture** creates a resource continuum through a peer-to-peer approach without central control points, management entities, or intrinsically privileged members. Following a decentralized model similar to the Internet, FLUIDOS fosters coexistence among various actors, from large cloud providers to territory-linked enterprises and small office/home owners.

- **Multi-ownership support** ensures each actor maintains full control over their infrastructure while deciding what resources and services to share and with whom. While individual clusters typically remain under single-entity control, the entire resource continuum spans different administrative domains.

- **Fluid topology** allows members to join or leave the virtual continuum at any time, regardless of infrastructure size, accommodating everything from enterprise-grade data centers to IoT and personal devices.

The FLUIDOS architecture consists of two main components: Nodes and Catalogs. While Nodes form the fundamental building blocks of the ecosystem, Catalogs are optional components that facilitate cross-domain interactions and provide user interfaces for public access [27].

4.4.1 FLUIDOS Node

A FLUIDOS Node represents a unique computing environment under the control of a single administrative entity. Multiple Nodes can exist under different administrative controls, allowing for a decentralized ecosystem. Each Node comprises one or more machines modeled with a common, extensible set of primitives that abstract the underlying details while exposing significant distinctive features.

In practical terms, a FLUIDOS Node is orchestrated by a single Kubernetes control plane and can consist of either a standalone device or a collection of devices such as those found in a datacenter. While device homogeneity simplifies management, it is not a requirement within a Node. Essentially, a FLUIDOS Node corresponds to a Kubernetes cluster extended with FLUIDOS-specific capabilities.

A FLUIDOS Node encapsulates a set of resources including computing, storage, networking, and accelerators, along with software services that can be utilized locally or shared with other nodes. Each Node features autonomous orchestration capabilities, accepting workload requests, executing jobs on administered resources when requirements and security policies are met, and maintaining a consistent set of policies for inter-node interactions.

4.4.2 FLUIDOS Supernode

A FLUIDOS Supernode serves as a gateway for domain nodes lacking direct Internet access or knowledge of other domains. While functioning similarly to standard nodes, Supernodes possess knowledge of other domains or catalogs for interaction. These interactions utilize the same protocols and components as regular node communications, though the sources and destinations of information differ [27].

Beyond their gateway function, Supernodes act as aggregation points in the information distribution chain. Their specific tasks are detailed in component documentation, while their interactions are evident in workflows spanning multiple domains.

4.5 Liqo

Liqo forms an integral part of the FLUIDOS ecosystem, providing the technical foundation for virtual cluster federation. Developed as an open-source project, Liqo enables dynamic and seamless resource sharing across multiple Kubernetes clusters, creating a unified virtual cluster that presents itself as a single entity to users [33].

4.5.1 Virtual Cluster Federation

Liqo's federation capabilities allow FLUIDOS to create virtual clusters that span physical boundaries. Through Liqo, a Kubernetes cluster can dynamically extend its capacity by leveraging resources from other clusters. This extension process is transparent to users and applications, which continue to interact with what appears to be a single cluster.

The federation process works through a peering mechanism that establishes trust relationships between clusters. Once peered, clusters can share resources based on configurable policies that define what resources are shared, with whom, and under what conditions. This capability is fundamental to FLUIDOS's goal of creating a fluid computing continuum that spans from edge to cloud.

4.5.2 Resource Offloading

Liqo implements a resource offloading mechanism that enables workloads to be scheduled on remote clusters. This process is handled transparently by extending the Kubernetes scheduler with awareness of remote resources. When local resources are insufficient or when specific capabilities are required that are only available in remote clusters, Liqo can offload workloads while maintaining their logical association with the home cluster.

4.6 FLUIDOS Node Architecture

The FLUIDOS node architecture builds upon Kubernetes, leveraging its ability to abstract underlying physical resources and capabilities in a uniform way regardless of whether dealing with single devices or full-fledged clusters. This foundation provides standard interfaces for resource consumption while extending Kubernetes with new control logic responsible for node-to-node interactions and enabling advanced policies and intents that are not natively supported by the orchestrator.

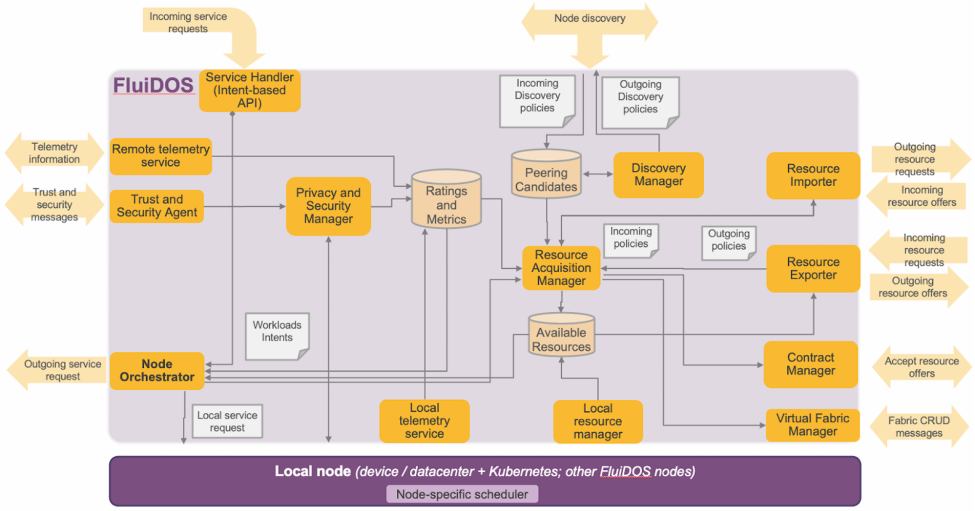

Figure 4.1: FLUIDOS node architecture

4.6.1 Core Components

The architecture revolves around two central elements: the Node Orchestrator and the Available Resources database. The Node Orchestrator manages service requests across local and remote nodes within the same fluid domain, coordinates interactions with both local components and remote nodes, and ensures services adhere to trust and security relationships. The Available Resources database maintains current information about resources and services available locally or acquired from remote nodes.

Beyond these central elements, several specialized modules extend the functionality of a FLUIDOS node:

Discovery Manager

The Discovery Manager is responsible for identifying other FLUIDOS nodes, maintaining a local database of feasible peering candidates. Each candidate is characterized by:

- A globally unique identifier
- Parameters necessary for peering and resource acquisition (e.g., network endpoints)
- Distinguishing features (geographical location, resource availability, hardware capabilities, software services)
- Optional pricing and billing models

These features, exposed through generic key-value labels, enable both policy-driven filtering and a priori ranking during resource acquisition. The Discovery Manager supports multiple approaches:

- FLUIDOS catalogs - directories of nodes that may be public or require authentication
- Multicast DNS - enabling seamless on-LAN clustering of independent devices
- Manual configuration - a fallback approach when other methods are unsuitable

The discovery process is governed by user-specified policies that determine whether the current node should be announced through certain approaches and which discovered peers should be accepted based on capabilities or trust levels.

Node Orchestrator

The Node Orchestrator coordinates service requests, determining whether they should be executed locally or offloaded to remote FLUIDOS nodes based on the Available Resources database. It interacts with the Resource Acquisition Manager to trigger acquisition of new resources when existing ones are insufficient.

Service requests are specified through an intent-driven API, describing desired outcomes rather than implementation details. Each request includes:

- Soft constraints (desired characteristics with priorities)
- Hard constraints (mandatory requirements)

These constraints are expressed in high-level service characteristics such as geographical location, maximum latency, specialized hardware needs, high availability guarantees, and cost constraints. The Node Orchestrator translates these intents into actual scheduling decisions with information from telemetry services.

When processing a request, the Node Orchestrator selects the target FLUIDOS node based on specified constraints by consulting the Available Resources database. If no suitable resources are found, it triggers the Resource Acquisition Manager to query peering candidates. Ultimately, it either schedules the workload locally or offloads it to a remote node through the remote service handler.

Resource Acquisition Manager

The Resource Acquisition Manager negotiates the acquisition of resources and services from remote FLUIDOS nodes. It can be triggered proactively based on policies to ensure future resource availability or reactively by the Node Orchestrator when matching resources are unavailable for a service request.

The process begins by consulting the peering candidates database, filtered and ranked based on service characteristics. The module then sends resource requests to selected candidates, specifying required capabilities and constraints. These requests are handled by the resource exporter module on the counterpart cluster, which may offer matching resources, a subset, or a superset based on availability and policies.

Different business models are supported, including:

- Reserved resources - dedicated to the requesting node and billed regardless of consumption
- Pay-per-use - charging only for resources actually consumed without guaranteeing continuous availability

Upon receiving offers, the Resource Acquisition Manager filters and ranks them based on user-defined policies, accepting those that best match requirements. The Contract Manager then formalizes the exchange, potentially using smart contracts, and adds acquired resources to the Available Resources database.

4.6.2 REAR - Resource Exchange And Registration

The REAR (Resource Exchange And Registration)[34] component serves as the unified interface through which FLUIDOS nodes exchange information about available resources and services. Acting as an intermediary layer between discovery protocols and the Node Orchestrator, REAR standardizes the resource advertisement and discovery processes regardless of the underlying protocol implementation.

REAR provides a consistent API for:

- Publishing node capabilities and available resources
- Discovering other nodes and their advertised resources
- Establishing initial communication channels for subsequent resource negotiations
- Maintaining up-to-date information about the resource landscape

This abstraction enables FLUIDOS to support multiple discovery mechanisms simultaneously, each optimized for different network environments while presenting a unified view to higher-level components. All discovered cluster information is stored in KnownCluster Custom Resource Definitions (CRDs), which include essential connection details such as IP addresses and ports for inter-cluster communication.

Virtual Fabric Manager

The Virtual Fabric Manager establishes the computing continuum abstractions that enable seamless execution of workloads across multiple nodes. Once a resource offer is accepted, it sets up the virtual node abstraction along with network and storage fabrics through interaction with the remote node.

This functionality is provided by Liqo, a project that extends Kubernetes abstractions to multi-cluster scenarios in three main directions:

- **Virtual Node:** Abstracts a remote FLUIDOS node as a local Kubernetes node, synchronizing necessary artifacts to enable seamless remote execution of unmodified workloads.
- **Network Fabric:** Ensures transparent and secure communication between workloads spread across multiple nodes, automatically managing potential conflicts such as overlapping address spaces.
- **Storage Fabric:** Enables a data continuum allowing workloads to attach to data lakes on the hosting node, following the data gravity approach to prevent expensive data migration and comply with regulatory requirements.

Privacy and Security Manager

The Privacy and Security Manager guarantees the security of all parties involved in the resource continuum. It works with the Trust and Security Agent, a component on each FLUIDOS node that certifies the correctness of predefined operations.

During node discovery, it ensures the trustworthiness of advertised features through appropriate proof mechanisms. For resource acquisition, it leverages a

rating and metrics database to filter and rank peering candidates based on previous experiences.

The manager ensures security from both perspectives:

- **Host perspective:** Guarantees that offloaded workloads do not harm the local system by confining related workloads in dedicated sandboxes with appropriate limitations on resource consumption, operations, and network connectivity. It also prevents side-channel attacks and supervises the provisioning of additional security mechanisms.
- **Guest perspective:** Provides guarantees that offloaded workloads are executed correctly without tampering and that telemetry data from hosting nodes is accurate. This information enriches the reputation value of specific nodes for future resource acquisition decisions.

The manager also handles Security Orchestration, dynamically selecting and configuring security services through interactions with the local orchestrator and telemetry services.

Telemetry Service

The Telemetry Service monitors infrastructure components, collecting observability parameters crucial for enforcing and verifying workload requirements. It consists of:

- **Local Telemetry Service:** Monitors the performance of the local node to determine locally available resources.
- **Remote Telemetry Service:** Exposes aggregated information about guest workloads to their original nodes, enabling monitoring of execution and verification of SLA compliance.

The Remote Trust and Security Agent plays a key role in preventing nodes from misrepresenting application performance. The collected monitoring data enriches the ratings and metrics database, influencing future resource acquisition decisions.

Cost Manager

The Cost Manager evaluates the burdens of computational workloads on nodes, considering both monetary and non-monetary factors. Three primary cost functions have been identified:

- Operational monetary costs

- Environmental costs, particularly carbon emissions
- Hardware production monetary costs

The cost function is designed to be generic and parameterizable to accommodate multiple cost types. The manager works closely with the local resource manager to monitor resources and sensors on machines and access external data such as the carbon intensity of local electricity grids.

4.7 Discovery Protocols

Central to the FLUIDOS architecture are the discovery protocols that enable nodes to find and connect with each other. These protocols form the primary subject of this thesis and are critical for establishing the dynamic, peer-to-peer network that underlies the FLUIDOS ecosystem.

The discovery protocols in FLUIDOS must operate across diverse network environments, including scenarios with limited connectivity, NAT traversal requirements, and varying levels of network quality. Two primary approaches have been implemented and evaluated:

4.7.1 Network Manager

The Network Manager implements a multicast-based discovery protocol that allows FLUIDOS nodes to announce their presence and capabilities to other nodes on the network. It leverages multicast for local network discovery. Its implementation sends announcement messages at 5s intervals, while garbage collection removes stale entries after 20s. The main limitation of this approach is its limitation to private network environments due to multicast restrictions on the public Internet.

Key features of the Network Manager include:

- Self-announcement through periodic multicast messages
- Capability advertisement including resource availability
- Automated peer management and garbage collection

4.7.2 Neuropil

Neuropil[35] implements a DHT-based discovery approach that provides a more scalable solution for large-scale deployments. Based on a distributed hash table architecture, Neuropil creates a structured overlay network based on the Tapestry DHT that enables efficient node discovery without relying on broadcast or multicast

mechanisms, allowing its operation across the public Internet and in churn scenarios. Furthermore, Neuropil provides end-to-end encryption for secure communication between nodes, ensuring data privacy and integrity.

Key features of Neuropil include:

- Decentralized node discovery without central coordination
- Structured overlay network for efficient routing
- End-to-end encryption for secure communication
- Resilience to network partitions and churn scenarios

Chapter 5

Evaluation Framework and Implementation

This chapter presents the design and implementation of the evaluation framework. The benchmarking tool is developed in Python, which leverages the Kubernetes Python client library to interact with the Kubernetes API.

5.1 Framework Design

A structured methodology was defined and followed in the design process:

1. Definition of specific use cases and their related requirements
2. Formulation of meaningful metrics for protocol evaluation
3. Development of benchmarking and automation tools
4. Collection and analysis of experimental data

As described in chapter 1, the primary FLUIDOS use case considered in this work involves offloading intensive computation from battery-powered UGVs to extend battery life while enabling more complex processing tasks. Vehicle-infrastructure communications represent another promising application area, potentially enabling autonomous driving, traffic management, and similar services. Both scenarios involve challenging network conditions, with robots or vehicles operating in remote areas, moving at high speeds or scenarios with many devices generating interference.

It is not possible to define specific figures that can be considered bad network conditions, as these can vary significantly depending on the specific use case. Research in connected vehicles offers some insights into 5G network performance,

but these are relevant for a specific setup-e.g 5G in motorway section[36]. For this reason, the evaluation considers different steps of network degradation, starting from the default network conditions and gradually increasing packet loss to 5%, 10%, and 25%, in order to test the behaviour of the discovery protocols under different scenarios. The final user of FLUIDOS, being aware of the network conditions in which their system will operate, can choose the most suitable protocol for their specific use case.

The metrics to be collected during the evaluation are as follows:

1. **Node discovery time**: Measurement of the time required for a device to discover another cluster

2. **Graceful exit time**: Evaluation of how long it takes for a device to intentionally leave the network and communicate its departure to other nodes

3. **Node failure detection time**: Assessment of how quickly the network detects when a node is no longer online (e.g., by tracking Time-To-Live expirations)

4. **Multi-tenancy capability**: Evaluation of each protocol's ability to:

   - Support multiple simultaneous network groups
   - Allow devices to belong to multiple groups concurrently
   - Manage cross-group discovery and communication

For computational offloading to occur, the client device must first discover an available cluster. From a user perspective, what matters is the time elapsed from starting the robot until it can begin offloading computation tasks. The evaluation therefore measures the interval between the activation of the discovery application pod and the receipt of information about an available cluster (specifically, its IP address and port).

The implementation begins with a minimal test configuration: direct communication between two computation clusters, each running a FLUIDOS node. This approach allows testing basic functionality before proceeding to more complex scenarios such as scalability testing. The testing process is automated by resetting conditions in Kubernetes to simulate a fresh FLUIDOS node starting, enabling statistical analysis of protocol behavior across multiple trials.

5.2 Testbed Setup

The testing environment consists of two Ubuntu 20.04 virtual machines, each configured with 4 vCPUs and 4GB RAM. While virtualization offers considerable flexibility, it also imposes certain networking limitations. For simplicity, network connectivity between the VMs is treated as a black box that allows them to communicate as if they were on the same Layer 2 network. The virtual machines run K3S instances as the orchestrator, selected for its low resource requirements [37].

The network configuration is particularly critical for the multicast-based Network Manager protocol. Standard Kubernetes pods receive a single virtual interface, but through the Multus meta-plugin[38], multiple interfaces can be assigned to a pod. Since Kubernetes networking does not natively support multicast, the NetworkManager implementation uses Multus to create a secondary interface running macvlan in bridge mode.

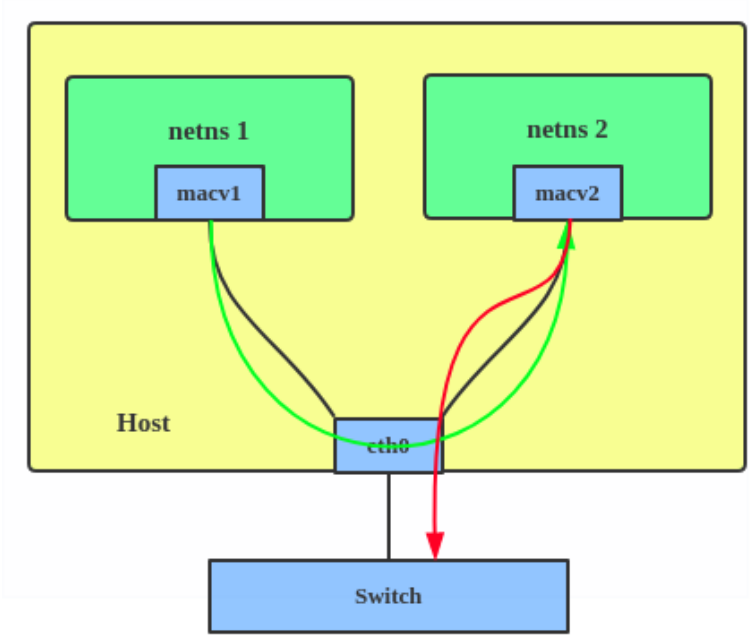

Figure 5.1: Scheme of the macvlan bridge mode [39]

Macvlan creates multiple interfaces with different Layer 2 MAC addresses on a single physical interface. In bridge mode, it connects all endpoints through the physical interface, as shown in fig. 5.1. This configuration enables multicast traffic to flow between FLUIDOS nodes within the same LAN.

Figure 5.2 illustrates the complete virtual machine setup used for testing.

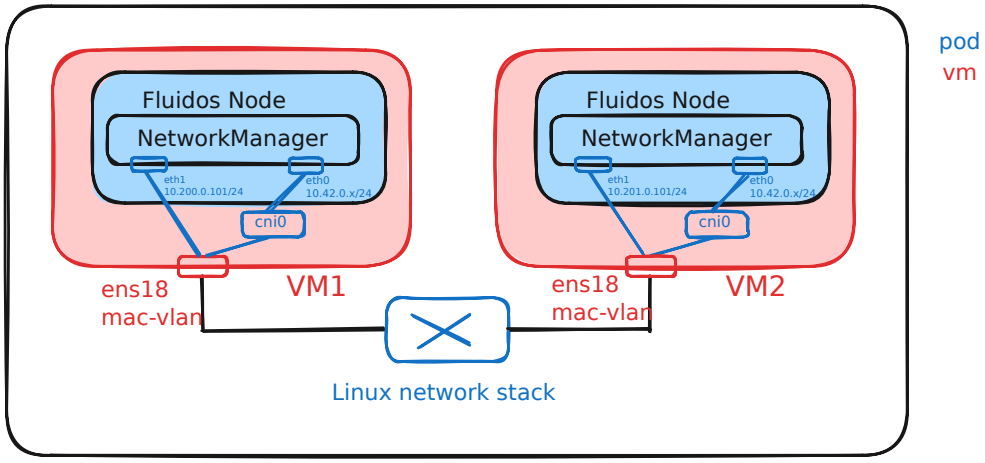

Figure 5.2: VM setup for testing

5.3 Relevant FLUIDOS Components and CRDs

In the FLUIDOS architecture, information about available clusters is stored in a Custom Resource Definition called `KnownCluster`. An example instance is shown below:

```
Name:         knowncluster-zygjltygdu
Namespace:    fluidos
Labels:       <none>
Annotations:  <none>
API Version:  network.fluidos.eu/v1alpha1
Kind:         KnownCluster
Metadata:
    Creation Timestamp:  2025-02-10T20:05:05Z
    Generation:          1
    Resource Version:    10951929
    UID:
   f9e9ff74-9df1-4d49-92b6-eb51c4aa98b6
Spec:
    Address:  172.18.0.8:30000
Status:
    Expiration Time:    2025-03-03T11:05:25Z
    Last Update Time:   2025-03-03T11:05:15Z
Events:              <none>
```

Listing 5.1: Example of a KnownCluster CRD instance

The key fields in this resource are `Address`, which contains the cluster's IP address and port for REAR, and the `Status` fields that implement a timeout mechanism through `Expiration Time` and `Last Update Time`.

For discovery services, FLUIDOS employs two different Kubernetes controller types: `DaemonSet` for the multicast-based NetworkManager and `Deployment` for the DHT-based Neuropil implementation.

A critical aspect of the implementation involves controlling pod count for testing purposes. This will be expanded upon in the next section

5.4 Benchmark Implementation

The benchmarking tool measures the time interval from discovery pod initialization until the detection of an available cluster (indicated by the creation of a KnownCluster resource). The benchmark process follows this sequence:

1. Scale down all discovery pods to zero
2. Delete existing KnownCluster resources and events to establish clean test conditions
3. Start the discovery pod and begin timing
4. Monitor for KnownCluster resource creation, then stop timing
5. Repeat multiple times to generate statistically significant results

While simply deleting the previous pod and allowing Kubernetes to recreate it might seem sufficient, this approach would not provide adequate control over the testing process. Instead, the implementation employs specialized patch operations for both resource types to ensure the pod count is set to zero while other operations, such as the deletion of other resources before the next test run, are executed.

For the `DaemonSet`, the node selector is modified with a non-existent label:

```
1 def disable_daemonset():
2     """Patch the DaemonSet to add a non-existent node label
    (disable it)."""
3     patch = [
4       {
5         "op": "add",
```

```
        "path": "/spec/template/spec/nodeSelector",
        "value": {"non-existent-label": "true"}
      }
    ]
    v1_daemonset.patch_namespaced_daemon_set(daemonset_name,
                                             namespace,
                                              patch)
    print("DaemonSet disabled (non-existent node label
   applied).")
```

Listing 5.2: Function to disable the daemonset

For the `Deployment`, the replica count is directly set to zero:

```
def disable_neuropil():
    """Scale down the Neuropil deployment to 0 replicas."""
    patch = [
        {
            "op": "replace",
            "path": "/spec/replicas",
            "value": 0
        }
    ]
    v1_deployment.patch_namespaced_deployment(np_deployment,
                                              namespace,
                                              patch)
    print("Neuropil deployment scaled down to 0 replicas.")
```

Listing 5.3: Function to disable the deployment

When initiating services for testing, similar functions perform the inverse operations. To capture timing data, the implementation uses Kubernetes' watch API to monitor for resource creation:

```
def watch_for_first_cr_creation(mode):
    """Watch for the creation of the first KnownClusters CR
       and measure the time it takes."""
    creation_time = None
    start_time = datetime.now()
    if mode == "netman":
        enable_daemonset()
    else:  # mode == "neuropil"
        enable_neuropil()
    w = watch.Watch()
```

```
11      print("Watching for the creation of the first
    KnownClusters CR...")
12
13      for event in
    w.stream(v1_custom.list_namespaced_custom_object,
14                               group=cr_api_group,
15                               version=cr_api_version,
16                               namespace=namespace,
17                               plural=cr_kind_plural):
18          if event['type'] == 'ADDED':
19              cr_name = event['object']['metadata']['name']
20              print(f"Detected creation of KnownClusters CR:
    {cr_name}")
21              creation_time = datetime.now() - start_time
22              w.stop()  # Stop the watch as we only need the
    first CR creation
23              break
24      F.write(f"{creation_time.total_seconds()}\n")
25      F.flush()
26      return creation_time
```

Listing 5.4: Function to watch for the creation of the KnownCluster CR

This process is repeated multiple times to generate statistically significant results.

5.5 Network Condition Emulation

Testing under default network conditions requires no modifications to the FLUIDOS node configuration. However, evaluating performance under degraded network scenarios necessitates additional implementation steps. The Linux Traffic Control (tc) command with the netem module is employed to simulate adverse network conditions, introducing controlled delay and packet loss.

The netem module affects outgoing packets from the interfaces to which it is applied. This means that in a two node setup, node A and node B, if the measures are taken on node A then the netem conditions have to be applied on the interface on node B. Applying these conditions requires direct access to container network interfaces, which presents a significant challenge: containers typically lack the necessary permissions to modify network settings due to security restrictions.

Since FLUIDOS nodes are deployed using Helm charts (a package management solution for Kubernetes applications), the `_helpers.tpl` file was modified to grant

the required permissions. The security constraints were removed and the container was given root access with `NET_ADMIN` capabilities:

```
...
{{/*
Get the Pod security context
*/}}
{{- define "fluidos.podSecurityContext" -}}
#runAsNonRoot: true
#runAsUser: 1000
#runAsGroup: 1000
#fsGroup: 1000
{{- end -}}

{{/*
Get the Container security context
*/}}
{{- define "fluidos.containerSecurityContext" -}}
runAsUser: 0       # Runs the container as the root user
runAsGroup: 0      # Runs the container as the root group
capabilities:
    add:
    - NET_ADMIN      # Grants the container permission to
   modify network interfaces
allowPrivilegeEscalation: true
{{- end -}}
```

Listing 5.5: Configuration to allow for network conditions modifications

To avoid manual configuration of network conditions for each test iteration, the pod manifest `fluidos-network-manager-daemonset.yaml` was modified to automatically configure the network when starting:

```
command: ["/bin/sh", "-c"]
args:
    - |
    apk add --no-cache iproute2 &&
    tc qdisc add dev eth1 root netem loss 10% &&
    /usr/bin/network-manager \
```

Listing 5.6: Modified manifest to automatically configure packet loss, in this case 10%

This approach enables automated testing under consistent adverse network

conditions, ensuring reproducibility and reliability across all experiments.

Chapter 6

Results

This chapter presents the experimental results obtained from benchmarking FLUIDOS discovery protocols under various network conditions and deployment scenarios. As mentioned in chapter 5, a two node VM setup was used for the initial testing of the protocols and compared with test runs obtained from physical machines, then the number of nodes was increased to evaluate the scalability of the protocols.

6.1 General results

Before presenting the results, some considerations regarding the evaluation metrics defined in section 5.1 are necessary.

Due to the current implementation of the protocols, some functionalities were not available.

- No graceful leave mechanisms are currently present in either protocol.
- In the case of Network Manager, the KnownCluster CRs are deleted after 20s, while in Neuropil no cleanup mechanism is present but should be implemented.
- No multi-tenancy support is present for Network Manager,but a workaround can be achieved by using defined multicast channels for each tenant, while Neuropil has authorization and authentication mechanisms in place.

With these considerations in mind, the following section present a deep dive into the analysis of the node discovery times for the two protocols under different network conditions.

6.2 Multicast-based results

The multicast-based discovery protocol, implemented through Network Manager, was tested first with the 2 VMs setup presented in fig. 5.2, consisting of two nodes that will be from now on referred as node A and node B. During testing node A has its discovery pod stopped and started as each test run is started, while node B remains running. The discovery time is measured as the time it takes for node A to discover node B.

Before going to the the figures and results, it is important to note that the implementation on Network Manager sends the multicast announce message every 5s. This is relevant to the results, as the implementation of the benchmark tool presented in chapter 5 aligns its measures to this time interval: the absolute first run of the tool is not considered in the results, as it starts at a random time with respect to the last received announce message. This applies to all the tests presented in this works, except for the scalability tests.

After receiving the announce message and creating the KnownCluster CR, the benchmark sends a request to Kubernetes to destroy the discovery pod and waits an arbitrary amount of time `sleep_time` by calling the `time.sleep(sleep_time)` function, before starting the next run. For the figures presented in this chapter, if not otherwise stated, the `sleep_time` was set to 2s.

The first test was conducted under default network conditions, with no packet loss. Figure 6.1 shows the distribution of discovery times for the multicast-based protocol run on the VM setup. The results show an almost normal distribution with a peak around 2.968s.

To verify if this behaviour depends on the deployment scenario, the same test was run on a bare-metal setup, consisting of two physical machines running a 4-core Intel N100 CPU, 16GB of DDR5 RAM and a 500GB SSD drive, connected to the same switch. From fig. 6.2, it can be seen that the distribution of discovery times is different compared from the one on the VM, showing a more uniform distribution with a 100ms difference between the minimum and maximum discovery time, while on the VM setup the difference is in the order of magnitude of tenths of milliseconds. This results is most probably due to virtualization, but the difference between the two setups is not deemed significant for the end user.

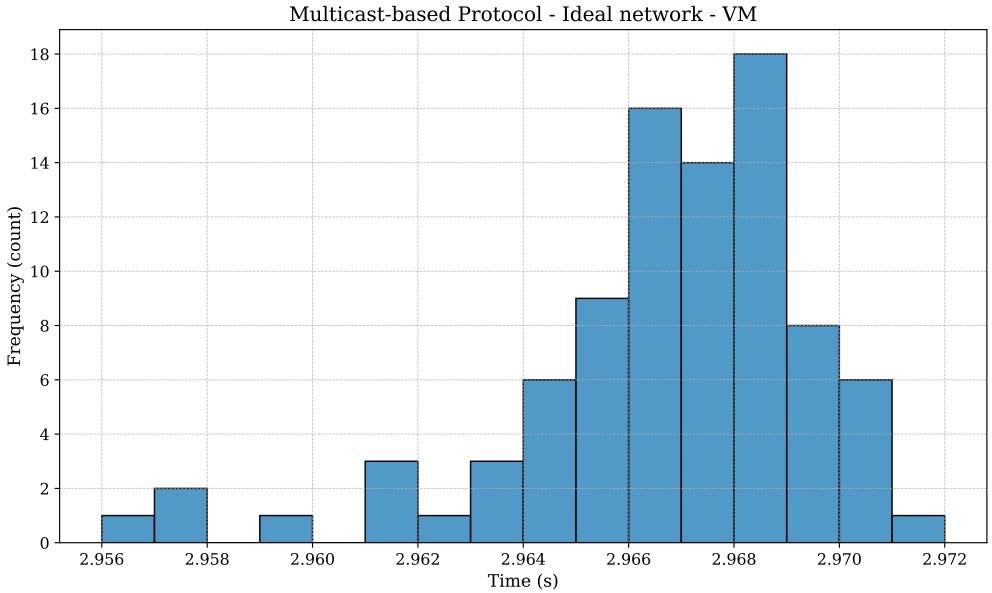

Figure 6.1: Discovery time distribution for multicast-based protocol under default network conditions (VM deployment)

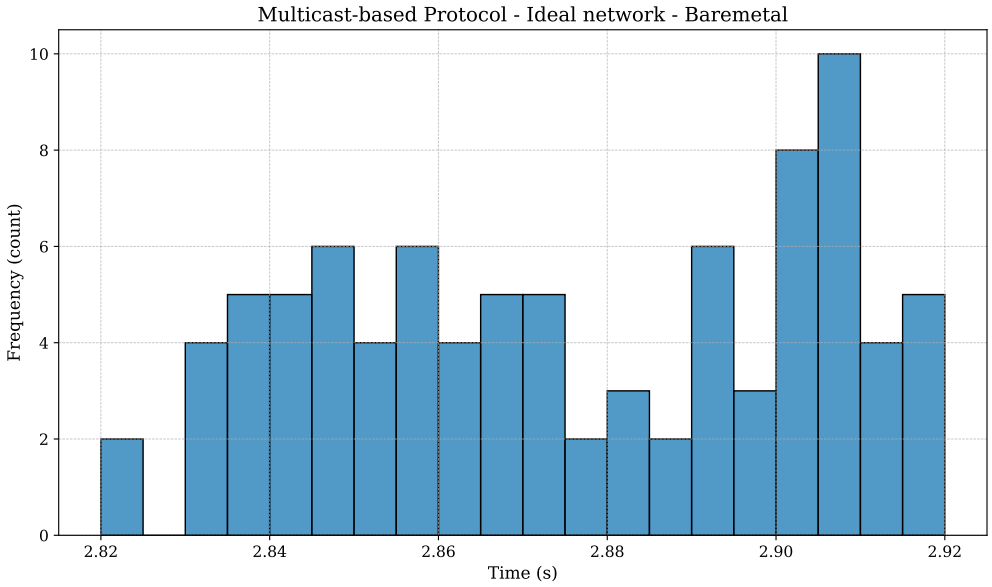

Figure 6.2: Discovery time distribution for multicast-based protocol under default network conditions (bare-metal deployment)

Having achieved a baseline for the multicast-based protocol, the next step is to

test it under bad networking conditions. The test was conducted with 5%, 10% and 25% packet loss. The last measure, despite being over the conditions limits discussed in the introduction of this thesis, is interesting to see the behavior of the protocol under extreme conditions. The impact of packet loss is expected to increase discovery times by 5s for each lost announce message. Naturally as packet loss increases, the more announce messages are lost and the longer the discovery time. The results are shown in figs. 6.6 to 6.8 for the tests run on the VM setup, while figs. 6.3 to 6.5, show the results obtained from the bare-metal setup.

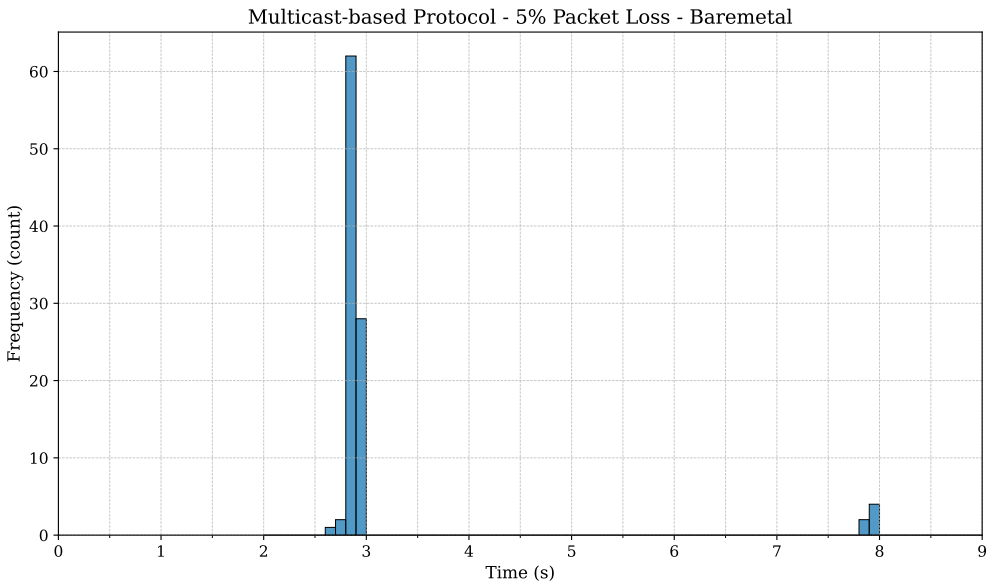

Figure 6.3: Discovery time with 5% packet loss (bare-metal deployment)

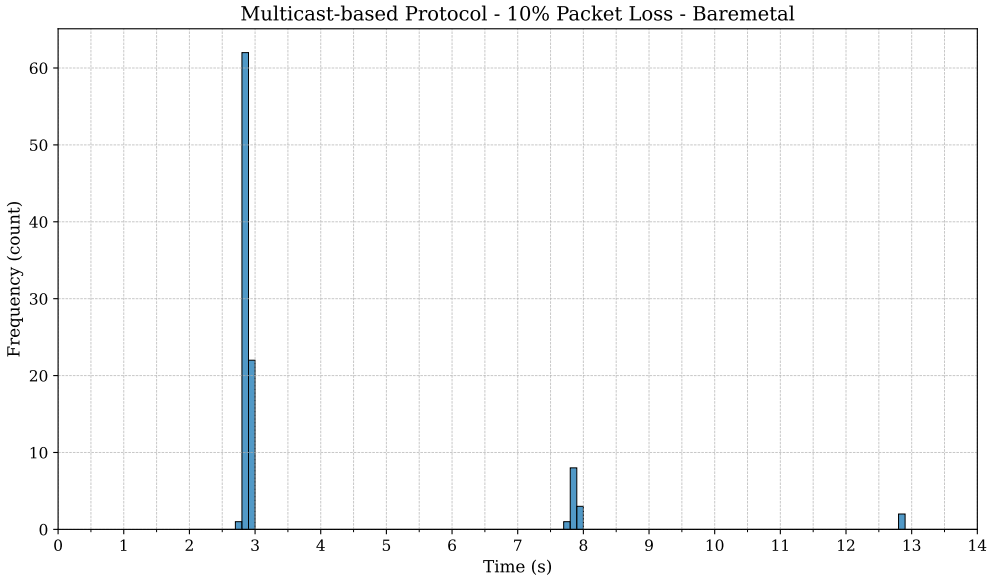

Figure 6.4: Discovery time with 10% packet loss (bare-metal deployment)

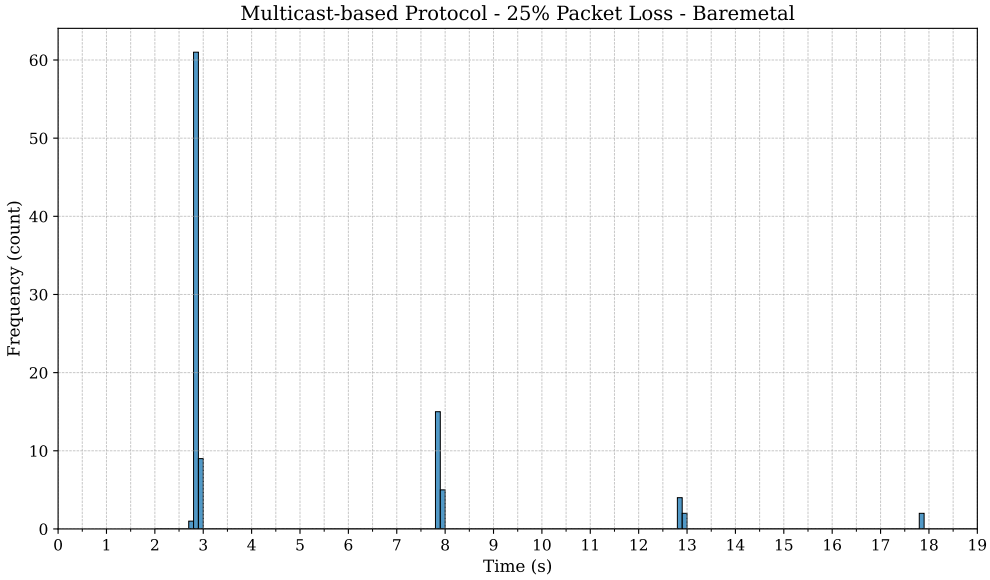

Figure 6.5: Discovery time with 25% packet loss (bare-metal deployment)

The results are in line with the expectations. As can be seen starting from fig. 6.3 the majority of samples cluster around 2.9s as per the baseline test, but some of the samples are instead clustered around 7.9s, which is the expected time increased by 5s meaning that in these cases the first announce message was lost

and was instead received at the second attempt.

This behaviour is more pronounced as the packet loss increases. From fig. 6.5 it can be seen as a higher percentage of samples cluster around $(2.9 + k \cdot 5)$s with $k \in \mathbb{N}$, and the worst samples in terms of discovery time achieve discovery at 18s.

Nonetheless, the protocol is still able to discover the other node, and more importantly, the discovery takes less than 20s. This is relevant as the implementation of Network Manager cleans up the KnownCluster CRs after 20s and if the protocol took more than this time there would be time intervals in which node A would not see node B as available.

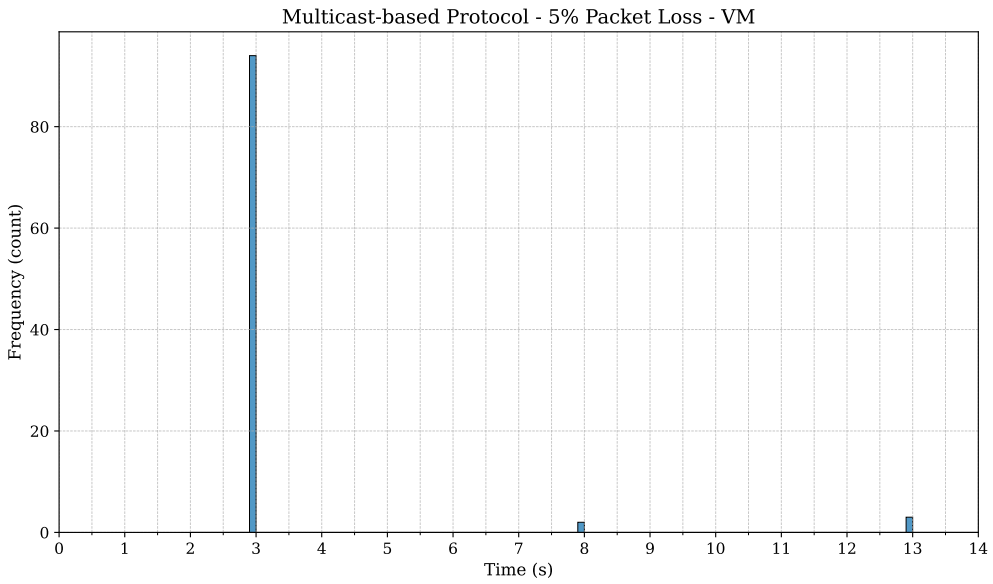

Figure 6.6: Discovery time distribution with 5% packet loss

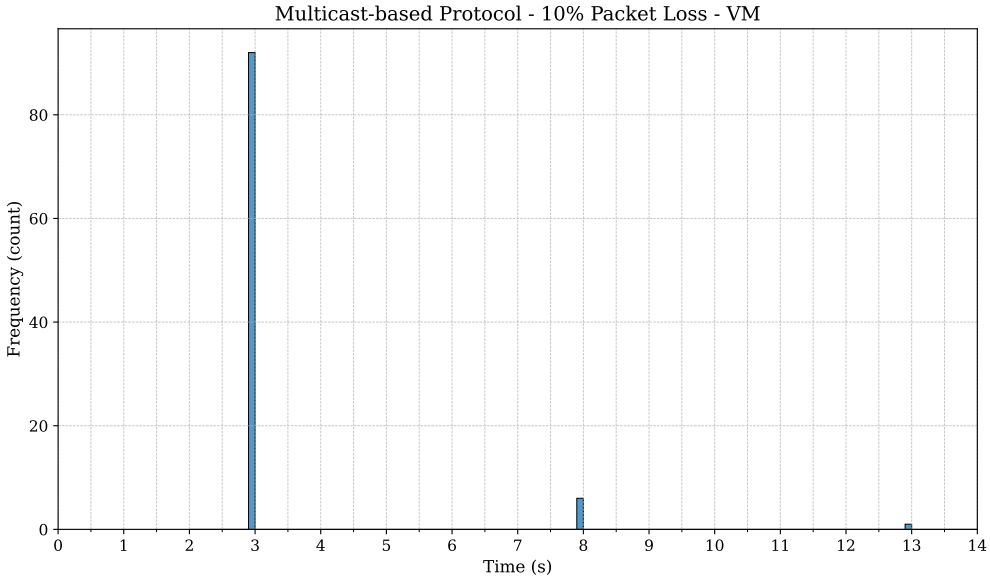

Figure 6.7: Discovery time with 10% packet loss

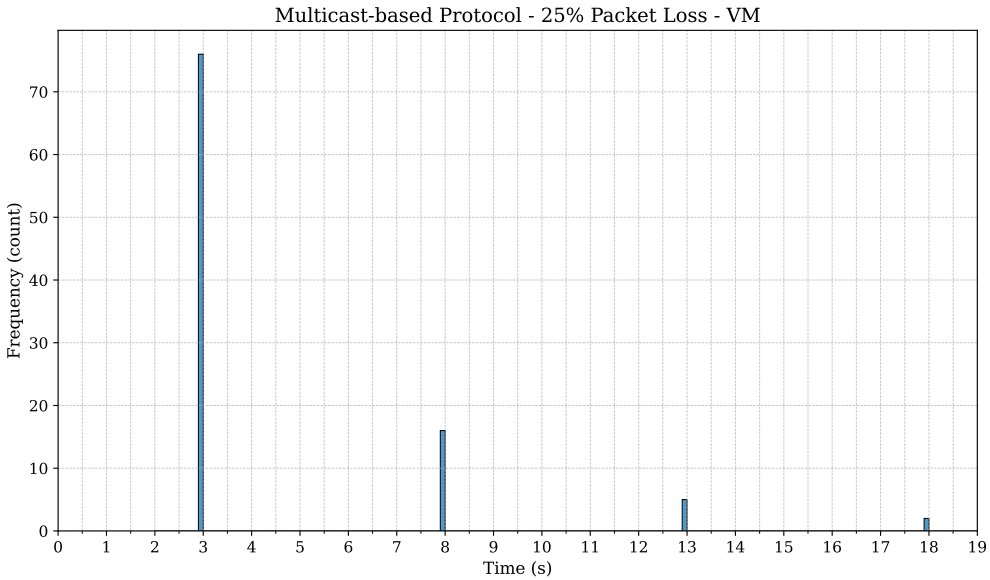

Figure 6.8: Discovery time with 25% packet loss

Test runs on the VM setup show similar results. The trend of clustering around baseline and increasing by 5s for each lost announce message is still present.

The differences with the physical machine results are the distribution of samples, which follows the results already seen in fig. 6.1, and the fact that in the 5% packet

loss test, the worst case scenario is 12.9s discovery time.

6.3 DHT-based results

The results for the DHT-based discovery protocol, implemented in Neuropil, follow the same structure as the multicast-based ones. Notice that in this discovery process the node joins an overlay network by bootstrapping to a node located in Germany via the public internet, with a latency of 25ms which remained constant for the durations of the tests.

An outcome of the testing is that the DHT-based protocol in its current implementation is not as reliable as the multicast-based one. Some difficulties were encountered during testing, such as a bug that did not allow automated testing to be run for more than 30 minutes, as node B would cease to appear online requiring its manual reset. From fig. 6.9 it can be seen that the baseline results have a considerably higher discovery time, with measures ranging from 35s and going up to more than 60s. As packet loss increases the discovery time increases as well as seen in fig. 6.10.

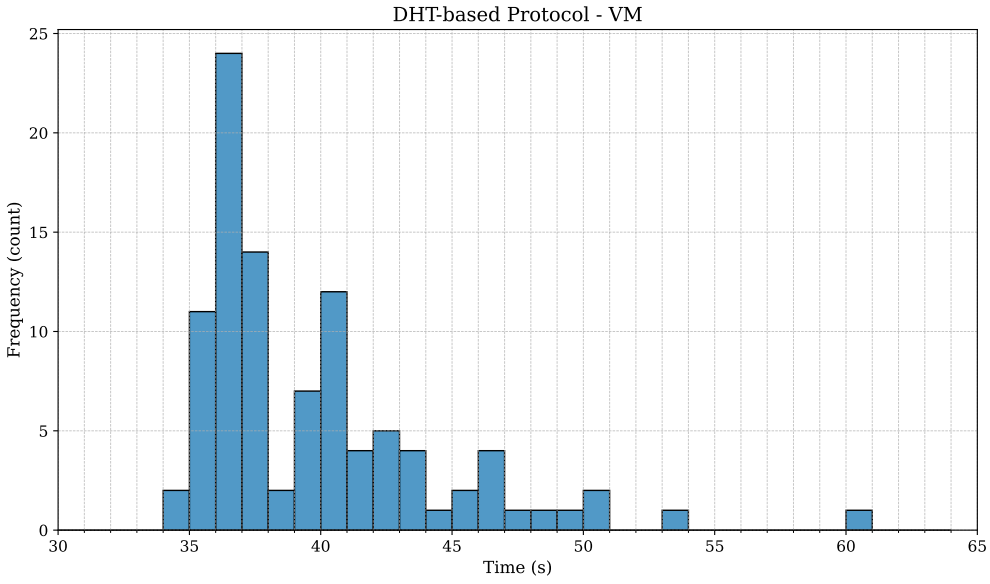

Figure 6.9: Discovery time distribution under default network conditions (VM deployment)

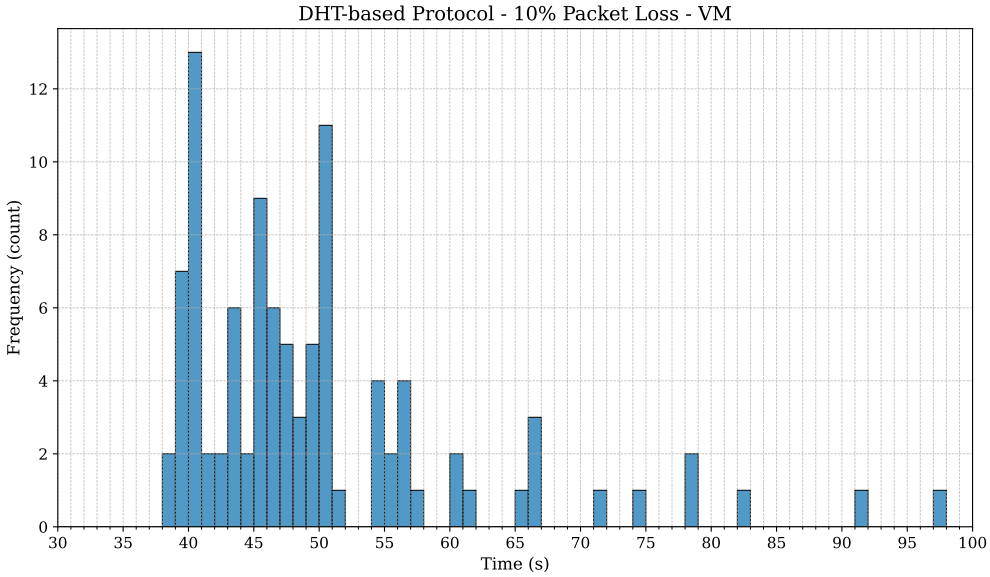

Figure 6.10: Discovery time distribution with 10% packet loss

6.4 Scalability results

After establishing the baseline for the two protocols and having tested them in a two node setup, the next step is the evaluation of their behaviour with a higher number of nodes.

Running multiple Kubernetes clusters requires a significant amount of resources, so the tests were run on a machine with Intel(R) Xeon(R) Gold 6442Y CPU with 96 logical cores, 512GB DDR5 RAM and 1.5TB SSD. To reduce the resource consumption, the clusters were run using Kubernetes in Docker (KinD) [40] which runs a containerized Kubernetes instance. The setup consists of 24 KinD clusters, so 24 FLUIDOS nodes, all connected to the same custom docker network, which behaves as a switch. In this setup, 23 remain always online, while 1 node called node A is started and stopped for each test run.

The scalability tests were run under default network conditions and with 5%, 10% and 25% packet loss. Each test run consists in gathering the time taken for the discovery of every new node, corresponding to the creation of a new KnownCluster CR, until every node is discovered. Since the number of discoverable nodes in a system with `N` nodes is `N-1`, the test run ends when 23 nodes are discovered. Every test consists of 100 test runs.

figs. 6.11 to 6.14 are line plots for 10 of the 100 runs. The aim of these plots is to show qualitatively the variability of the discovery times, while statistical results

will be provided later in this section. It can be noticed that each individual run has some inconsistencies in the discovery times, as a particular run that discovers 10 nodes before another does not necessarily discover all the other 23 nodes first.

In default network conditions, it can be qualitatively observed from fig. 6.11 that the discovery time distribution remains almost constant for each number of nodes discovered. This cannot be said for bad network conditions, as seen in figs. 6.12 to 6.14 where especially from 20 nodes discovered onwards, the discovery time variation increases considerably.

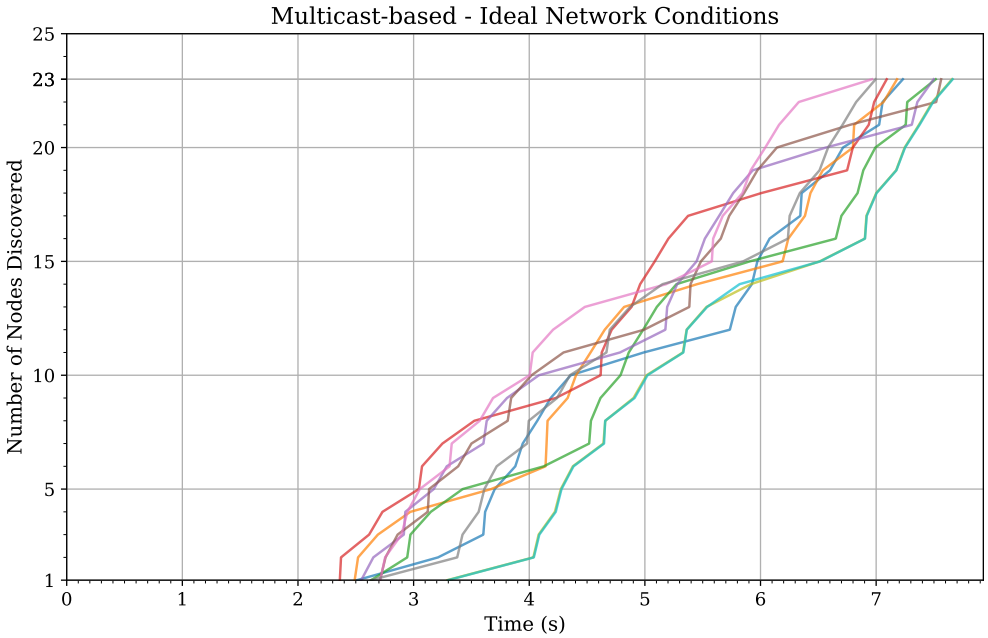

Figure 6.11: Network Manager scalability under default network conditions

figs. 6.15 to 6.17 show the scalability results for the DHT-based protocol, Neuropil. Before commenting the results, notice that bootstrap node located in Germany is setup in such a way that only 14 nodes are present in its routing table, limiting the number of nodes that can be discovered with this test setup. The results show that the current implementation of Neuropil is not yet ready to be released at scale. fig. 6.15 shows that out of 10 runs, none were able to find all other nodes. Furthermore, as time progresses, the muximum number of nodes discovered each run decreases. fig. 6.16 shows the same trend, but in this case out of 10 runs, 3 of them did not find any other node. Having observed that the protocol currently does not behave as intended, these test should be repeated in order to obtain data that is statistically significant.

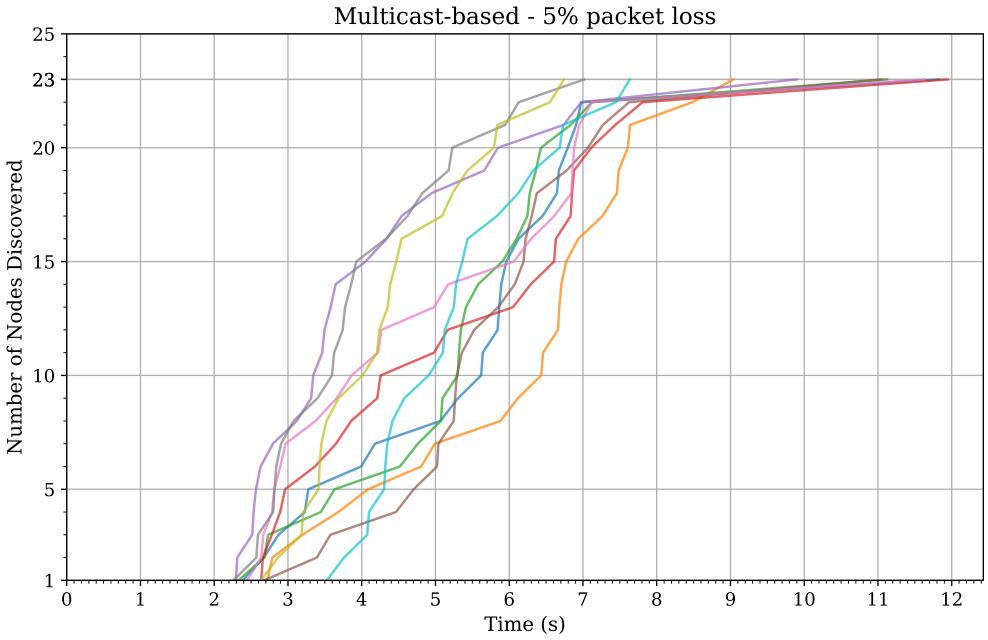

Figure 6.12: Network Manager scalability with 5% packet loss

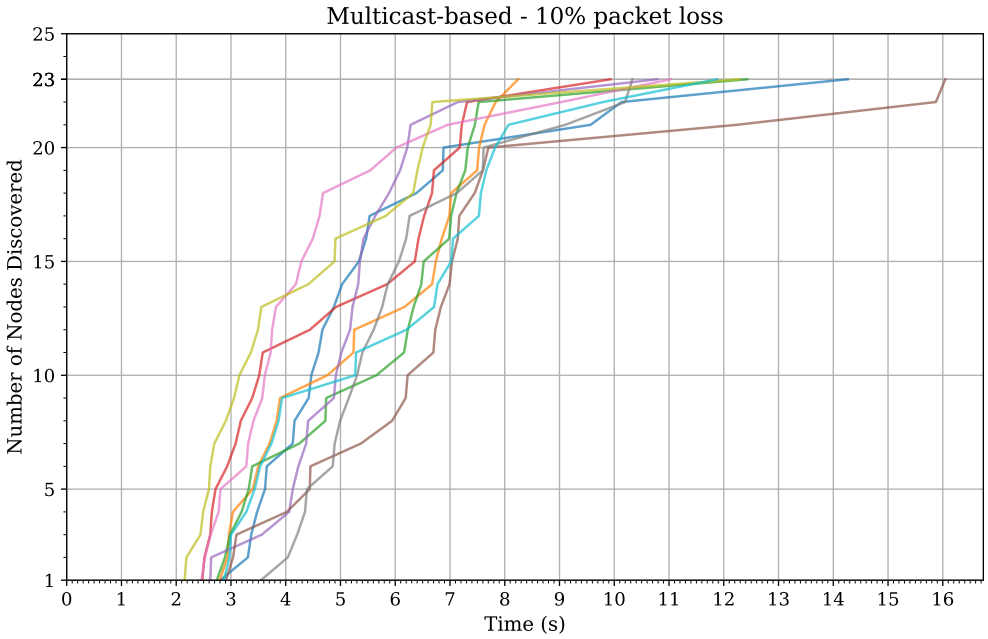

Figure 6.13: Network Manager scalability with 10% packet loss

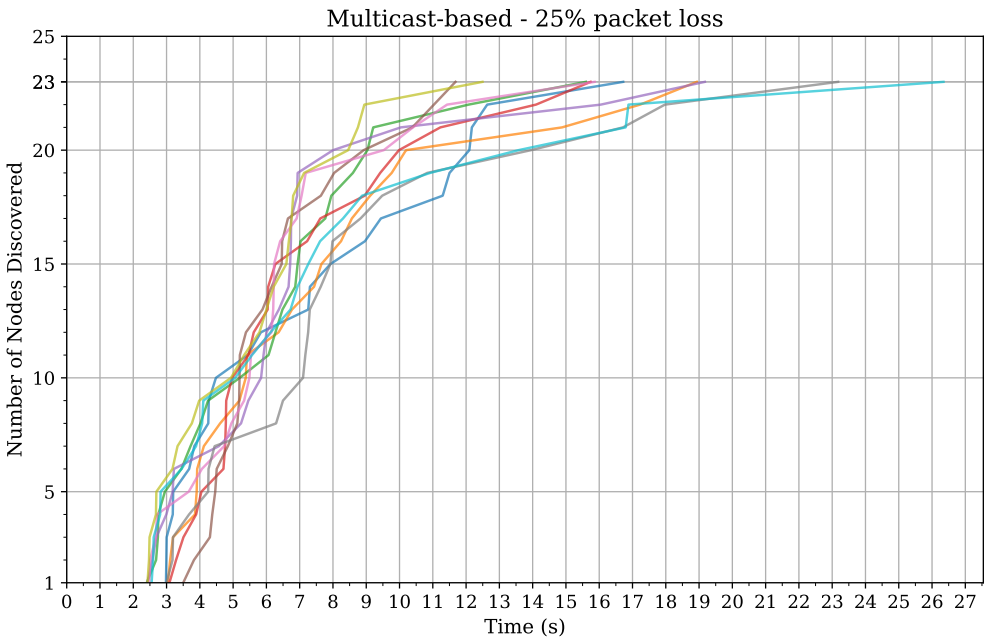

Figure 6.14: Network Manager scalability with 25% packet loss

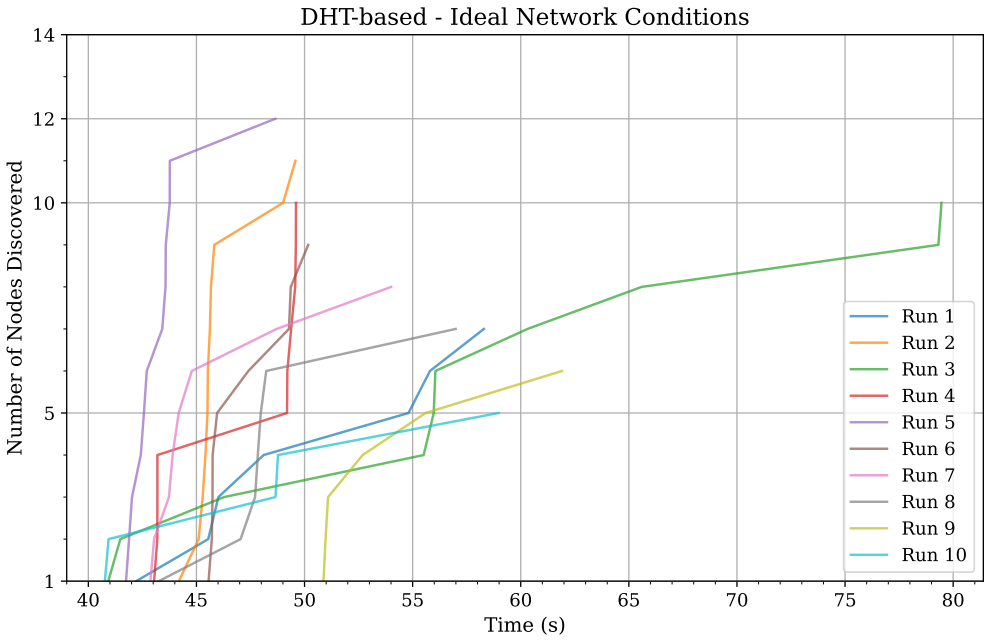

Figure 6.15: Neuropil scalability under default network conditions

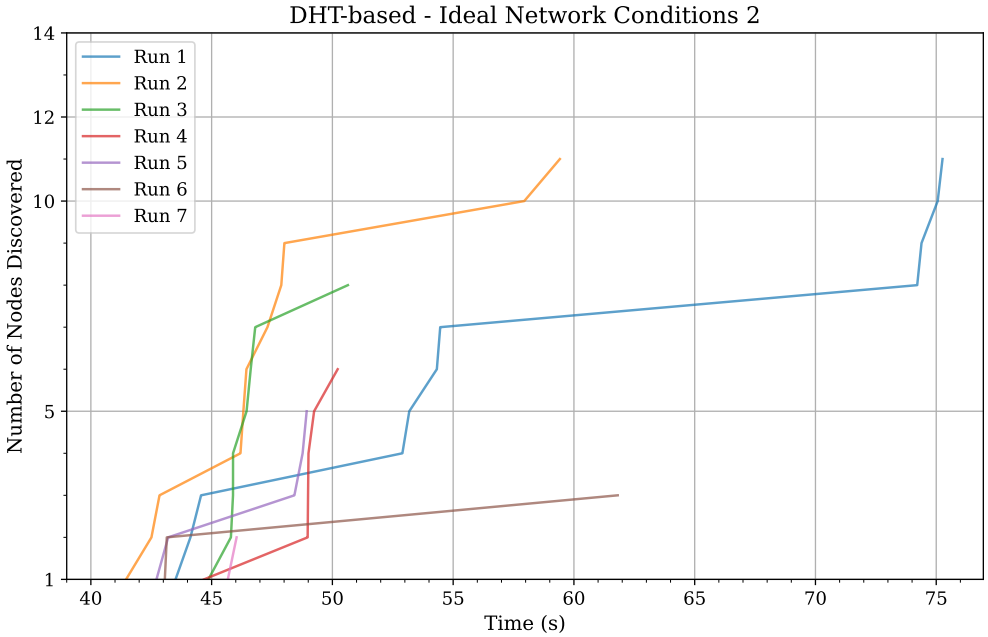

Figure 6.16: Alternative view of Neuropil scalability under default conditions

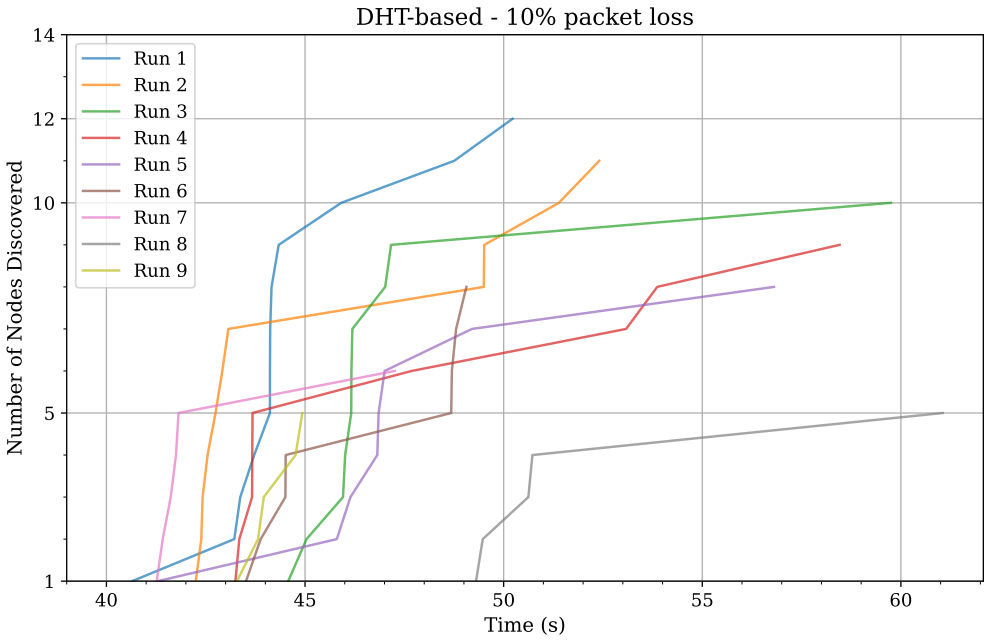

Figure 6.17: Neuropil scalability with 10% packet loss

Figures 6.18 to 6.21 show the box plot analysis of the Network Manager scalability tests. Results for Neuropil could not be gathered due to the current implementation of the protocol not allowing for automated testing of its behaviour at scale.

As seen in fig. 6.18, in ideal network conditions the distribution of node discovery time per number of nodes discovered remains approximately constant. As packet loss increases, the distribution of discovery times becomes more variable, as shown in figs. 6.19 and 6.20 , where the interquartile range increases for the same number of nodes discovered. For example, considering 10 nodes discovered, the interquartile range in ideal conditions is approximately half compared to the 5% packet loss case.

Notice that in the 25% packet loss case, shown in fig. 6.21, this behaviour is not as pronounced for low numbers of nodes discovered and for some cases the interquartile range is even lower compared to the 5% and 10% packet loss cases. It is instead more pronounce for higher numbers of nodes discovered, considering in this case from 18 nodes discovered onwards.

Considering the median values of discovery time for the number of nodes discovered, it can be seen that the median discovery time increases with higher packet loss. Analyzing the difference between the median discovery time for 23 nodes and 1 node, in fig. 6.18 it can be observed to be approximately $(7.5-3)s \simeq 4.5s$ for ideal conditions, in fig. 6.19 $(10-2.5)s \simeq 7.5s$ for 5% packet loss, $(11-2.5)s \simeq 8.5s$

for 10% packet loss and finally $(16-3)s \simeq 13s$ in the case of 25% packet loss as shown in figs. 6.20 and 6.21.

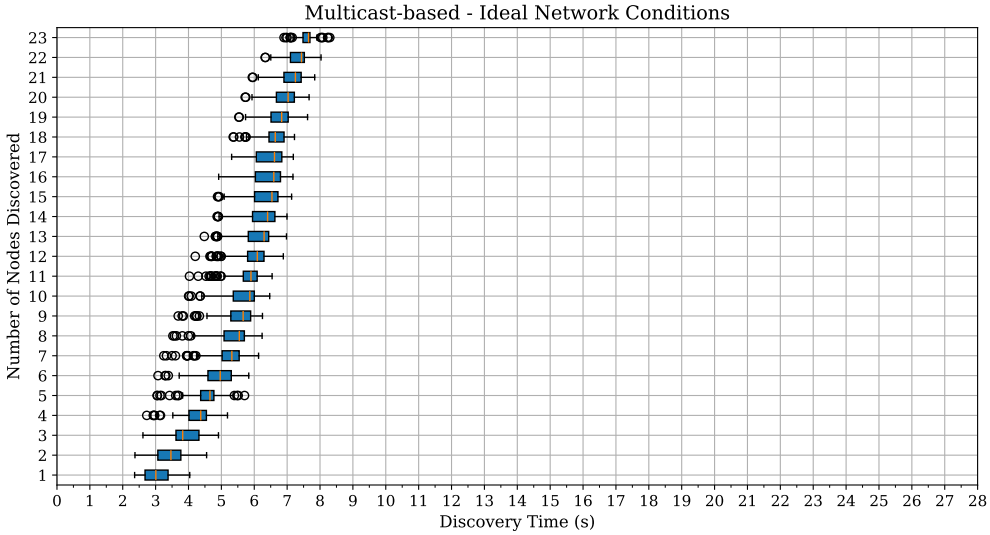

Figure 6.18: Box plot analysis of Network Manager scalability under default conditions

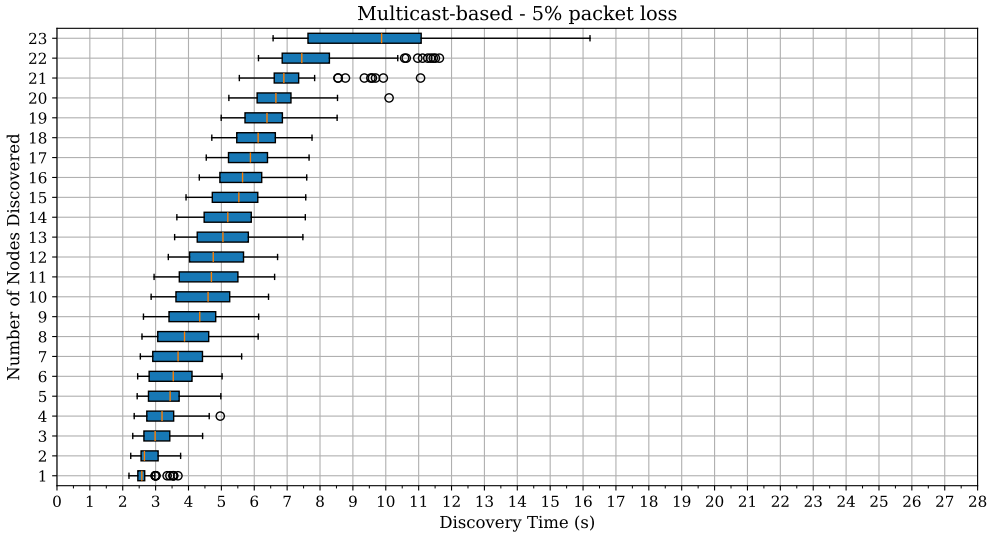

Figure 6.19: Box plot analysis of Network Manager scalability with 5% packet loss

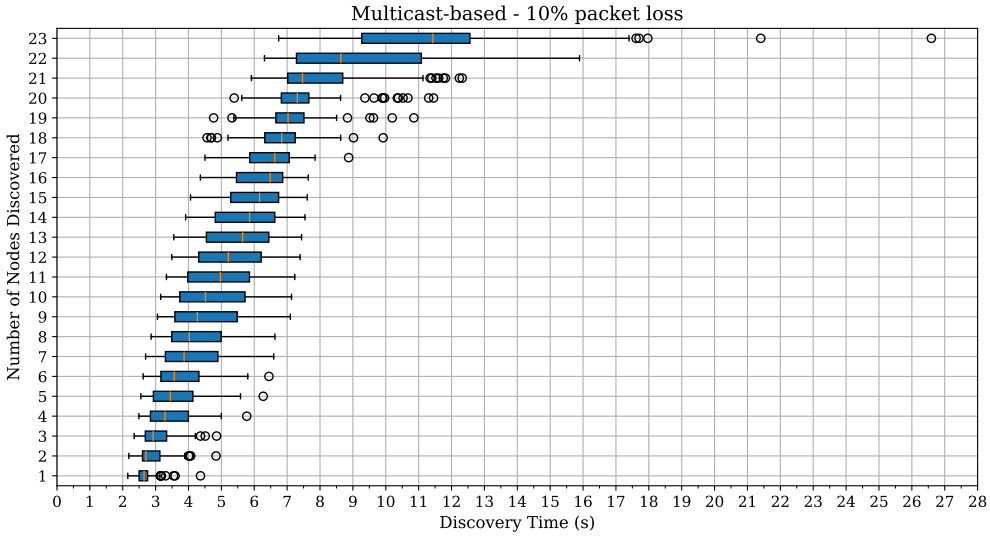

Figure 6.20: Box plot analysis of Network Manager scalability with 10% packet loss

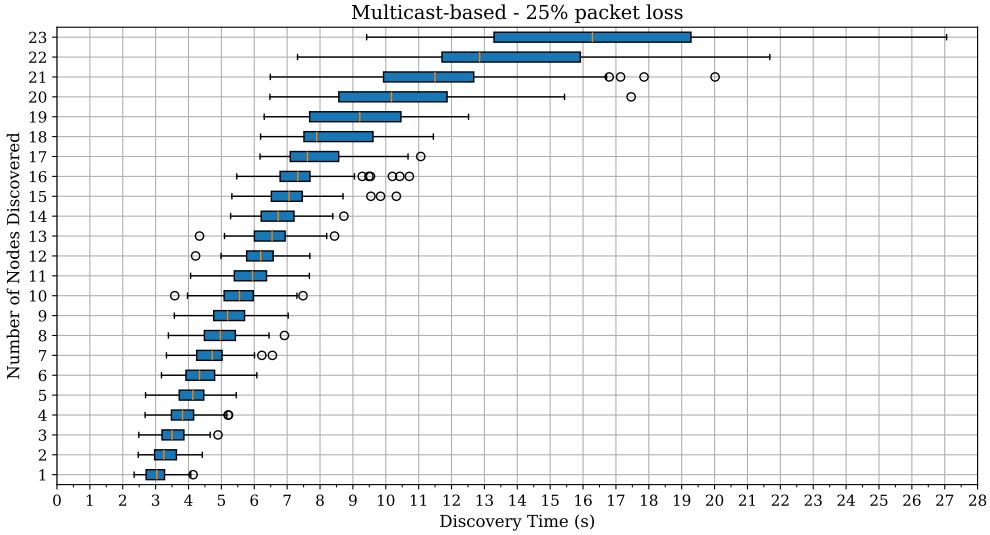

Figure 6.21: Box plot analysis of Network Manager scalability with 25% packet loss

Chapter 7

Conclusions

This thesis has presented a comprehensive evaluation framework for discovery protocols in the FLUIDOS ecosystem, focusing on comparing the performance of a multicast-based protocol (Network Manager) and a DHT-based protocol (Neuropil) under various network conditions and deployment scenarios.

The experimental results demonstrate that the multicast-based protocol provides reliable and consistent discovery performance across both virtual machine and bare-metal deployments. Under ideal network conditions, the discovery time averaged around 2.9 seconds, with minimal variation between deployment scenarios. As packet loss increased from 5% to 25%, the protocol exhibited a predictable degradation pattern, with discovery times increasing by approximately 5 seconds for each lost announce message, but maintaining functionality even under extreme conditions.

In contrast, the DHT-based protocol showed significantly higher baseline discovery times (35-60 seconds) and less consistent performance. The protocol exhibited reliability issues during extended testing periods, requiring manual intervention to maintain functionality. More critically, scalability testing revealed that the current implementation of Neuropil is not yet ready for large-scale deployment, with inconsistent node discovery capabilities and declining performance in multi-node environments. Nonetheless, the protocol demonstrated potential for improvement, allowing for discovery of nodes spanning international geographical locations.

Scalability analysis of the multicast-based protocol demonstrated its ability to maintain relatively consistent discovery times as the network expanded to 24 nodes. The statistical box plot analysis revealed that while the interquartile range of discovery times increased with higher packet loss, the protocol continued to function effectively across all tested conditions.

The evaluation framework developed for this study provided a structured approach to benchmarking discovery protocols in FLUIDOS, enabling automated

testing and providing statistical analysis of significant performance metrics. Furthermore, this work will provide from now on a replicable testing methodology that can be extended to evaluate future discovery protocols and improvements to existing ones, such as Neuropil.

7.1 Future Work

Based on the findings and limitations of this study, several directions for future work can be identified. First of all, a re-evaluation of the DHT-based protocol would be beneficial in a successive release to allow for a more comprehensive analysis of its performance and reliability. Moreover, the testing setup could be extended to include more complex network topologies, higher node counts and different networking scenarios such as high node churn in the network. Finally, the evaluation framework could be extended to include additional performance metrics such as network overhead and energy consumption.

Bibliography

[1] Eric Brewer. «CAP twelve years later: How the "rules" have changed». In: *Computer* 45.2 (2012), pp. 23–29 (cit. on pp. 15, 25).

[2] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. «Conflict-Free Replicated Data Types». In: *Stabilization, Safety, and Security of Distributed Systems*. Springer. 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29. URL: https://link.springer.com/chapter/10.1007/978-3-642-24550-3_29 (cit. on p. 17).

[3] Nuno Preguiça. *Conflict-free Replicated Data Types: An Overview*. 2018. arXiv: 1806.10254 [cs.DC]. URL: https://arxiv.org/abs/1806.10254 (cit. on p. 20).

[4] Jim Bauwens and Elisa Gonzalez Boix. «Memory efficient CRDTs in dynamic environments». In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. VMIL 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 48–57. ISBN: 9781450369879. DOI: 10.1145/3358504.3361231. URL: https://doi.org/10.1145/3358504.3361231 (cit. on pp. 20, 34, 35, 37).

[5] António Barreto, Hervé Paulino, João A. Silva, and Nuno Preguiça. «PS-CRDTs: CRDTs in highly volatile environments». In: *Future Generation Computer Systems* 141 (2023), pp. 755–767. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.12.013. URL: https://www.sciencedirect.com/science/article/pii/S0167739X22004186 (cit. on pp. 20, 34).

[6] David Karger, Eric Lehman, Tom Leighton, Rik Panigrahy, Matthew Levine, and Daniel Lewin. «Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web». In: (1997), pp. 654–663 (cit. on pp. 21, 24).

[7] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. «Tapestry: a resilient global-scale overlay for service deployment». In: *IEEE Journal on Selected Areas in Communications* 22.1 (2004), pp. 41–53. DOI: 10.1109/JSAC.2003.818784 (cit. on pp. 21, 22, 34, 35).

[8] Zhonghong Ou, Erkki Harjula, Otso Kassinen, and Mika Ylianttila. «Performance evaluation of a Kademlia-based communication-oriented P2P system under churn». In: *Computer Networks* 54.4 (2010), pp. 689–705 (cit. on pp. 23, 34, 35).

[9] Avinash Lakshman and Prashant Malik. «Cassandra: A decentralized structured storage system». In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40 (cit. on p. 23).

[10] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. «Epidemic algorithms for replicated database maintenance». In: *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*. 1987, pp. 1–12 (cit. on p. 24).

[11] Apache Software Foundation. *Cassandra Hardware Choices*. n.d. URL: https://cassandra.apache.org/doc/5.0/cassandra/managing/operating/hardware.html (cit. on pp. 25, 34–36).

[12] U.S.P. Srinivas Aditya, Roshan Singh, Pranav Kumar Singh, and Anshuman Kalla. «A Survey on Blockchain in Robotics: Issues, Opportunities, Challenges and Future Directions». In: *Journal of Network and Computer Applications* 196 (2021), p. 103245. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2021.103245. URL: https://www.sciencedirect.com/science/article/pii/S1084804521002435 (cit. on pp. 26, 27, 36).

[13] Uber Engineering. *How Uber Optimized Cassandra Operations At Scale*. n.d. URL: https://www.uber.com/en-IT/blog/how-uber-optimized-cassandra-operations-at-scale/ (cit. on p. 34).

[14] Basho Technologies. *Riak*. n.d. URL: https://riak.com (cit. on p. 35).

[15] Alex Khawalid, Dan Acristinii, Hans van Toor, and Eduardo Castelló Ferrer. «Grex: A Decentralized Hive Mind». In: *Ledger* 4 (Apr. 2019). DOI: 10.5195/ledger.2019.176. URL: https://ledger.pitt.edu/ojs/ledger/article/view/176 (cit. on p. 36).

[16] Moritz Platt, Johannes Sedlmeir, Daniel Platt, Jiahua Xu, Paolo Tasca, Nikhil Vadgama, and Juan Ignacio Ibanez. «The Energy Footprint of Blockchain Consensus Mechanisms Beyond Proof-of-Work». In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, Dec. 2021. DOI: 10.1109/qrs-c55045.2021.00168. URL: http://dx.doi.org/10.1109/QRS-C55045.2021.00168 (cit. on p. 36).

[17] L. Baird, M. Harmon, and P. Madsen. *Hedera: A Governing Council & Public Hashgraph Network, The Trust Layer of the Internet*. Whitepaper. Version 2.1. Accessed: 2021-11-07. 2021. URL: https://hedera.com/hh_whitepaper_v2.1-20200815.pdf (cit. on p. 36).

[18] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. «Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade». In: *Queue* 14.1 (2016), pp. 70–93 (cit. on p. 38).

[19] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: The Complete Guide to Master Kubernetes*. O'Reilly Media, 2017 (cit. on p. 39).

[20] Randy Bias, Thomas A Limoncelli, and Christina J Hogan. *The Practice of Cloud System Administration: DevOps and SRE Practices for Web Services*. Vol. 2. Addison-Wesley Professional, 2016 (cit. on p. 39).

[21] Gigi Sayfan. *Mastering Kubernetes: Level up your container orchestration skills with Kubernetes to build, run, secure, and observe large-scale distributed apps*. Packt Publishing, 2021 (cit. on pp. 39, 45, 54).

[22] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, 2021 (cit. on pp. 39, 40, 43, 54).

[23] Marko Luksa. *Kubernetes in Action*. Manning Publications, 2018 (cit. on pp. 40, 42, 43, 54).

[24] Kubernetes. *Kubernetes API*. 2023. URL: `https://kubernetes.io/docs/concepts/overview/kubernetes-api/` (cit. on p. 42).

[25] Kubernetes. *Cluster Networking*. 2023. URL: `https://kubernetes.io/docs/concepts/cluster-administration/networking/` (cit. on p. 44).

[26] Kubernetes. *Custom Resources*. 2023. URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/` (cit. on p. 46).

[27] FLUIDOS Project. *FLUIDOS Public Deliverables, D2.1 Scenarios, Requirements and Reference Architecture – V.1*. 2023. URL: `https://fluidos.eu/public-deliverables/` (cit. on pp. 47, 51, 53, 55, 56).

[28] Project Calico. *Calico Open Source: Networking and Security for Containers, VMs, and Native Host Workloads*. 2023. URL: `https://www.projectcalico.org/` (cit. on p. 48).

[29] CoreOS. *Flannel: A Network Fabric for Containers*. 2023. URL: `https://github.com/flannel-io/flannel` (cit. on p. 48).

[30] Isovalent. *Cilium: eBPF-based Networking, Security, and Observability*. 2023. URL: `https://cilium.io/` (cit. on p. 48).

[31] Blesson Varghese, Rajkumar Buyya, Nalini Subramanian, and Erol Kendall. «Computing across the continuum: Challenges and opportunities». In: *IEEE Internet Computing* 25.6 (2021), pp. 25–35 (cit. on p. 51).

[32] Yuang Liu, Cheng Yang, Li Jiang, Shengli Xie, and Yan Zhang. «Edge computing for the Internet of Things: A case study». In: *IEEE Internet of Things Journal* 6.3 (2019), pp. 4187–4198 (cit. on p. 54).

[33] Liqo Project. *Liqo: Dynamic and Seamless Kubernetes Multi-Cluster*. 2023. URL: `https://liqo.io/` (cit. on p. 56).

[34] FLUIDOS Project. *REAR: REsource Allocation and Ranking*. 2023. URL: `https://github.com/fluidos-project/REAR` (cit. on p. 59).

[35] pi-lar GmbH. *Neuropil*. URL: `https://www.neuropil.org/tutorial/` (cit. on p. 62).

[36] Dániel Ficzere, Gábor Soós, Pál Varga, and Zsolt Szalay. «Real-life V2X Measurement Results for 5G NSA Performance on a High-speed Motorway». In: *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2021, pp. 836–841 (cit. on p. 65).

[37] Rancher Labs. *K3s Requirements*. n.d. URL: `https://docs.k3s.io/installation/requirements?_highlight=requ` (cit. on p. 66).

[38] Intel Corporation. *Multus CNI: Enabling Multiple Network Interfaces for Pods in Kubernetes*. 2023. URL: `https://github.com/k8snetworkplumbingwg/multus-cni` (cit. on p. 66).

[39] Red Hat. *Introduction to Linux interfaces for virtual networking*. 2018. URL: `https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking#ipvlan` (cit. on p. 66).

[40] Kubernetes. *Kubernetes in Docker*. n.d. URL: `https://kind.sigs.k8s.io/` (cit. on p. 81).