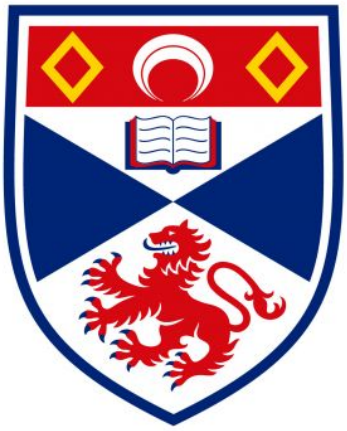
Lab-Based ILNP Emulation in Rust

Jordi Bugler-Lamb - 180005383

January 16, 2025

The School of Computer Science

Supervisor: Saleem Bhatti



University of
St Andrews

Since 1413

1 Abstract

This project provides a lab-based emulation of the Identifier-Locator Network Protocol (ILNP) over IPv6 multicast, that can be used to experiment with the ILNP addressing architecture without any change to the kernel or middleware network systems. ILNP separates identity and location functions in the network stack, to improve multi-homing and mobility, ensuring seamless connectivity. The report contains the emulation design, the deployment instructions, and qualitative analysis and evaluation of the application.

2 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 17,311 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

3 Introduction

The internet has become an important part of our daily lives. However, the original designs of the internet were under the assumptions that end-systems would be connected to a single network in fixed locations. This assumption has been challenged with the growing number of end-systems connecting to multiple networks (e.g. 4G and Wi-Fi on mobile phones) and switching networks as the device travels physically[1]. Various solutions have been proposed to provide these functionalities while limiting the performance degradation. Among these, the Identifier-Locator Network Protocol (ILNP) has emerged, which separates the identity and location functions in the network stack. This allows mobile systems to be multi-homed and move across networks while limiting disruption to the end-to-end communication flow [2]. ILNP outperforms other solutions, excelling in performance and enabling incremental deployment, making it easier to adopt over time[3]. This is because ILNP doesn't require proxies or middle-boxes, tunnels, address translation functions, or large-scale updates to core infrastructure. ILNP does however require changes made to the end-system kernels for adoption. Some researchers may not have the privileges to modify their system's kernel.

The aim of this project is to build a lab-based emulation of ILNP on IPv6 multicast. This will allow experimentation with the ILNP addressing architecture, without the overhead of setting up OS kernel installations. Distinct connected networks will be simulated using different IPv6 multicast groups. End-systems, connected to these simulated networks, will use the Identifier-Locator Network Protocol (ILNP) to route between the networks and exchange information.

Primary Objectives	**O1**	Create a network emulation of ILNP on lab machines
	O2	The emulated network should be implemented as an overlay network on top of Multicast IPv6.
	O3	The software should allow users to set up custom networks with custom traffic flows between these networks.
	O4	The user should be able to send traffic as intended through this emulated network using an API that in turn uses the ILNP API using custom nodes.
Secondary Objectives	**O5**	The software should allow users to switch their node's network while a node is transmitting data. As the flow uses ILNP the connection should not break.
	O6	Provide enough documentation for the user to make use of the protocol's output and be able to build applications on top of this protocol.
Tertiary Objectives	**O7**	Allow users to use more reliable protocols such as TCP
	O8	Create an interface to help the user create an experiment in a user-friendly way

3.1 Mobile Nodes on the Internet

Protocol Layer	IP (IPv4 and IPv6)	ILNP (ILNPv6)
Application	FQDN / IP address	FQDN
Transport	IP address	Identifier (NID)
Network	IP address	Locator (L64)
Physical i/f	IP address	Mac address

Figure 1: Names in IP vs ILNP [4].

The network layer, transport layer and the application layer all use the IP address to maintain the state of the packet flow[3] (as seen in Figure 1). Weiser [5] explains that a device with an IP address such as 13.2.0.4 is expected to reside in network 13, with all devices using IP addresses of the form 13.x.x.x also assumed to be within the same network. This assumption breaks down when a device switches to another network such as network 36 (Stanford). Simply changing the device's IP address is not a viable solution, as TCP assumes the IP address remains constant during the life of an end-to-end connection. Current versions of TCP cannot distinguish between packet loss due to mobility hand-offs and packet loss due to network congestion [6]. This inability leads to a

significant decrease in throughput and increase in delays during hand-offs. The issue becomes evident in scenarios such as when mobile devices switch from home Wi-Fi networks to cellular networks, resulting in an IP address change that degrades data flow performance or even terminates the connection. To address this limitation, applications have had to implement their own mechanism to handle IP address changes. This was the case for *IPv4* and *IPv6* [3] until solutions were implemented in the network stack. This background allowed us to understand some problems ILNP is trying to solve.

3.2 Contributions of this Paper

In this paper, we present an application as an additional layer on top of our current network stack. Our network stack is composed of UDP at the transport layer, IPv6 at the network layer and gigabit Ethernet at the link layer [7] to conform with objective 2. Multiple instances of the application can be deployed on the same machine or on separate machines. When running the application on separate machines, no changes are required to the current physical network such as routers or end-systems. This matches the ILNP requirements as no changes are required to the middleware, and conforms to the software requirements as no changes are required to the OS installation. For objective 1, 3 and 4, the application provides the user with a set of API to send and receive packets using the Identifier-Locator Network Protocol across the user's customised simulated network. In summary, the paper provides the user with:

- A customisable network topology using simulated nodes and routers running over multicast IPv6 (O2, O3).
- An API to send information between the simulated nodes and routers using ILNP (01, 04).
- A Protocol Control Block to analyse the performance of the chosen topology.

3.3 Structure of this Paper

In the Context Survey (section 4), we will cover background research that has already been carried out to improve multi-homing and mobility on the internet. This will strengthen our knowledge of the problem, and help identify key design decisions when implementing our solution. We will then outline the application's requirements (section 5). We will briefly discuss the software engineering process and ethics in section 6 and 7 respectively. In the design section (section 8), we will include design decisions that helped us meet the software requirements. In implementation (section 9), we will discuss the structure of application's code, implementation decisions, and we will provide deployment instructions. Performance testing (section 10) will test how well we have met the basic requirements. The Evaluation and Critical Appraisal section (section 11) will analyse the application in a real life agriculture scenario and critique any performance issues. We will also discuss how the project can be extended. Finally, the conclusion (section 12) will confirm what objectives have been completed, the quality of the solution and discuss how the application can be improved.

4 Context Survey

The context survey explores the evolution of internet technologies aimed at addressing challenges associated with hand-offs for mobile end-systems. It begins by examining solutions implemented at the application level followed by those integrated in the network stack. By studying each approach, we aim to understand the challenges, the progression of solutions, and their limitations, with the goal of optimising our design. ILNP is in the project requirements, however the context survey will also highlight why the Identifier-Locator Network Protocol (ILNP) is a valid candidate for hand-offs and mobility.

4.1 Session Initiation Protocol (SIP)

The Session Initiation Protocol (SIP) is responsible for creating sessions, inviting and removing devices from the sessions, and keeping track of the device's locations on the network. The protocol could be extended to support audio and video conferencing [8]. This allowed devices to maintain sessions during IP address changes. Upon leaving a network to join another, a User Agent notices the local IP Address has changed and sends a *RE-INVITE* request to the SIP server that in turns sends a *RE-INVITE* to the host server. The host server then updates its session parameters such as the mobile device's destination IP Address. This can be seen in Figure 2. SIP, however, operates at the application layer, meaning each signal sent also includes the transport protocol's overhead. As a result, the protocol does not scale well and demands significant computational resources. SIP has higher delays and hand-off disruption times compared to network level solutions [9]. SIP lacks security and typically requires TLS, which further increases computational load[10]. This background research helped understand the need for a solution at the IP layer. In our implementation, we separate signaling from the transport protocol to reduce overhead and improve scalability.

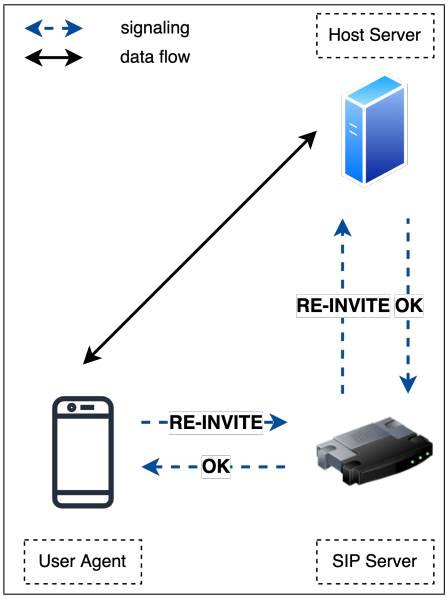


Figure 2: Session Initiation Protocol maintaining a session

4.2 Mobile IPv4

IP Mobility Support for IPv4 was one of the earliest attempts to solve mobility at the network layer [11]. We discuss Mobile IPv4 (MIP) as it sets the stage for more advanced protocols like ILNP. We discuss its limitations and why the need of a better solution was necessary. MIP uses a Home Agent to create a connection with the Remote Host that the Mobile Node is trying to connect to. Initially the Mobile Node is home and uses the home router with standard NAT. Upon leaving the house and connecting to the cellular network a tunnel is initiated between the Home Agent and the Foreign Agent. The Foreign Agent is in the network the Mobile Node has moved to. Traffic originating from the Remote Host is sent to the Home Agent to maintain the connection's IP address and then redirected to the Foreign Agent through the tunnel. The Foreign Agent can then respond directly to the Remote Host by replacing its source IP with the Home Agent's IP to pretend the packet came from the Home Agent[11]. This fools the application in thinking there's a single continuous flow of information. However, in reality there's a triangular communication. Information is sent from the Remote Host to the Home Agent, then using a tunnel from the Home Agent to the Foreign Agent and finally from the Foreign Agent to the Remote Host again creating triangular routing (as seen in Figure 3). This breaks the end-to-end principle [12].

Mobile IPv4 introduces many issues[13]. First, some firewalls block packets with incorrect source address which will affect the Foreign Agent in the foreign network. Secondly the tunneled packets might get blocked by firewalls using Deep Packet Inspection (DPI). Another issue is security: if for instance the Foreign Agent is not authenticated, attackers can offer mobility services to visiting Mobile Nodes and snoop on the packets being forwarded[13]. Furthermore, protocols such as TCP assume the channel is symmetric when using algorithms such as congestion control[11]. A triangular communication is asymmetric. Tunneling also increases packet size which can cause packet fragmentation[2]. There are many more issues [13].

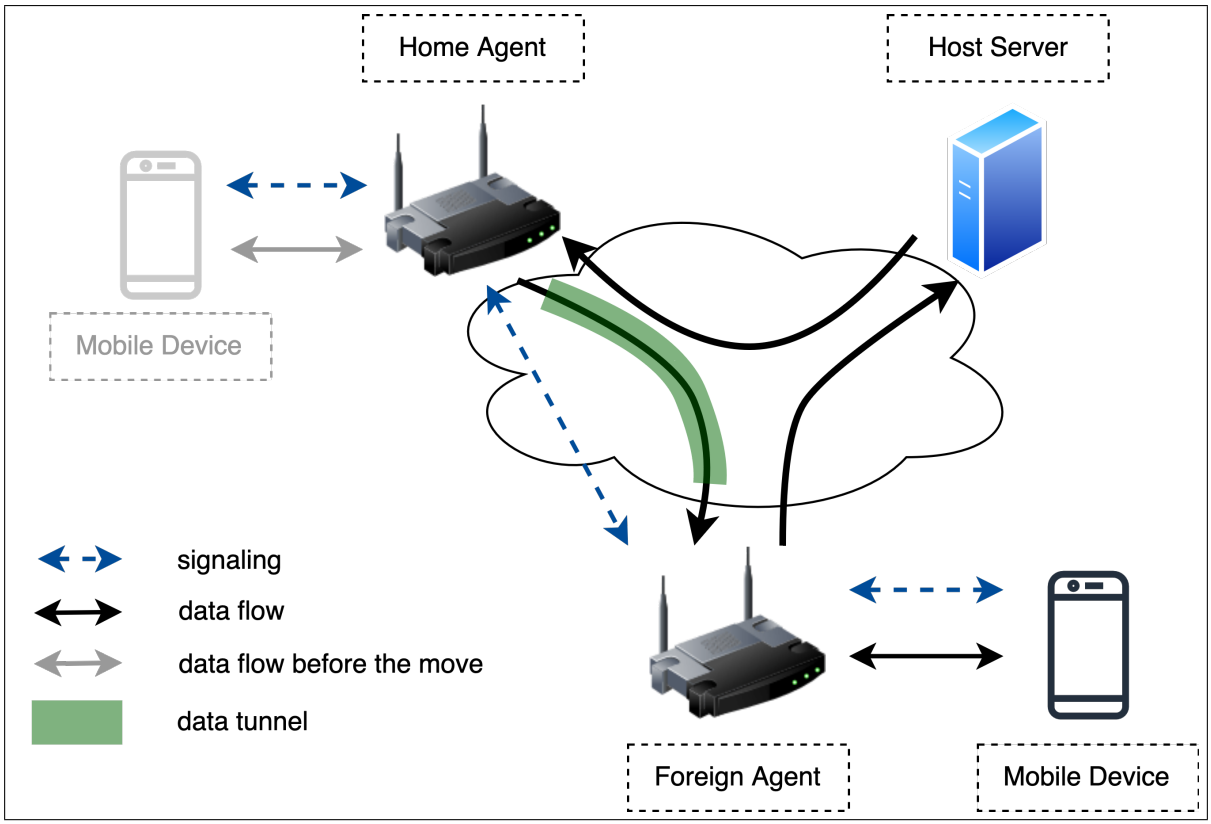


Figure 3: Mobilev4 triangular routing

4.3 Mobile IPv6

Mobile IPv6 (MIPv6) [14] was developed to address some of these issues by removing the use of a Foreign Agent and allowing the Mobile Node to notify the Home Agent and the Remote Host it has changed IP address (Care of Address, CoA) using Binding Update messages. This allows the Mobile Node to determine the best route with the Remote Host and removes triangular communications. The connection is now end-to-end and, we can make use of IPSEC[15] to secure the connection. The use of MIPv6, unfortunately, still has other issues. We still need to make use of the Home Agent and tunneling when route optimisation is not available. We also need to upgrade middleware systems in the network making it hard to deploy incrementally[3]. Furthermore, MIPv6 performs a hard handover, also known as *"break before make"*. This means during the hand off, latency and loss are introduced into the packet flow[3]. Finally, when choosing a new CoA, we need to perform Duplicate Address Detection (DAD) to check the new address is not already in use which can introduce additional delays [2]. We discussed Mobile IPv6 because it was the proposed solution at the time ILNP was written. IETF were discussing solutions to improve MIPv6 [2].

4.4 Locator/ID Separation Protocol (LISP)

Traditional IP addresses provide two functionalities, identifying the device and locating the device in the network topology. The Locator/ID Separation Protocol (LISP) separates these two functionalities to enable mobility and scalability. We discuss this protocol, as at the time of writing this report, LISP is a proposed alternative to ILNP. The Endpoint Identifier (EID) identifies a device on the network and remains constant, even as the device moves across different networks. The Routing Locator (RLOC) specifies the device's current location on the network. This changes as devices move across the network. LISP encapsulates packets with an outer IP header containing the RLOC. This allows LISP routers to use the outer IP header to forward packets accordingly [16]. Figure 4 demonstrates a device moving from the *RLOC3* to the *RLOC4* network. The LISP server updates its map to link the device's *EID* to its new *RLOC* address. When the host server sends a packet to the device, the packet is encapsulated with an IP header containing the *RLOC4* IP address. Upon arriving in the right network the packet is decapsulated and the *EID* is used to forward the packet to the right device. LISP is a good alternative to Mobile IPv6,

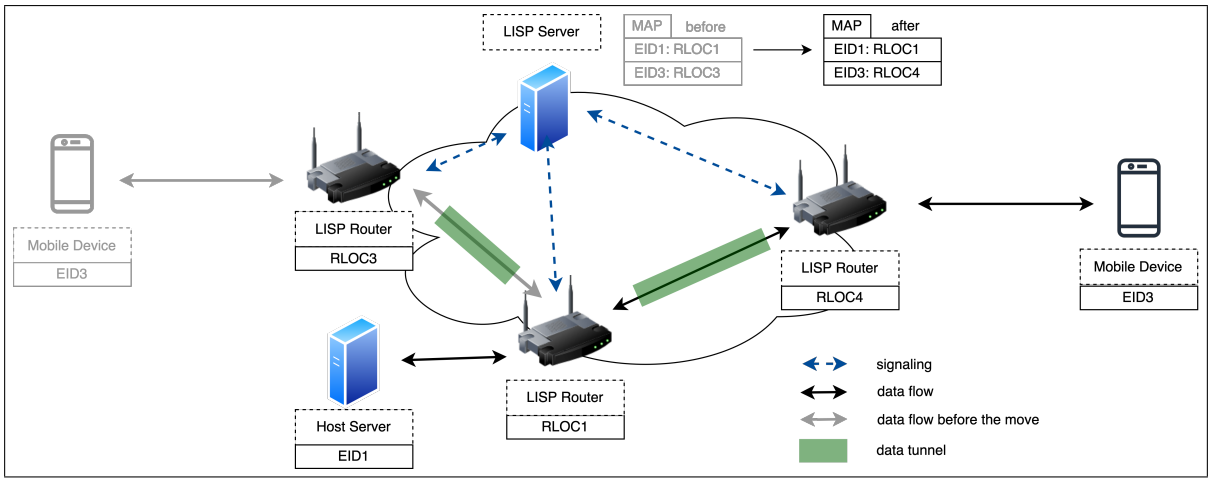


Figure 4: LISP device switching networks and using a new path to transmit data

it offers seamless multi-homing and mobility. Devices can switch between connections without disrupting the packet flow, using a "make before break" approach. However, LISP requires LISP compatible routers to be deployed across the network making it hard to deploy incrementally [16]. It requires tunneling which increases the packet size and can lead to additional packet fragmentation. Finally, LISP relies on a third-party server to keep a mapping of the *EID*s to *RLOC*s. This introduces additional packet overhead due to signaling and can introduce additional delays as we wait to resolve *EID*s to *RLOC*s.

4.5 Identifier-Locator Network Protocol (ILNP)

ILNP was introduced as an alternative to Mobile IPv6 to provide mobility with multi-homing, NAT and Security [2]. ILNPv6 is a superset of IPv6 with additions described in RFC6743[17] and RFC6744[18]. It eliminates the need for proxies, middle boxes and tunneling [19]. Changes are made to the end-systems and not middleware making it easier to deploy incrementally. ILNP uses the top 64-bit of a 128-bit IPv6 address as the locator (*L64*) and the lower 64-bit as the node's identifier (*NID*) [2]. Because ILNP splits the original IPv6 address to provide identity and location, we also do not require encapsulation. The locator is dynamic and changes as the mobile node moves across the network. Routers will use this to forward packets to the mobile node's network. The *NID* remains static and maintains the end-to-end state at the transport and application layer (as seen in Figure 1).

$P : port \quad A : ip\_address \quad N : nid \quad L : locator \quad X : source \quad Y : destination$
(1) $<tcp : PX, PY, AX, AY><ip : AX, AY><if : AX>$
(2) $<tcp : PX, PY, NX, NY><ilnp : (LX), (LY)><if : (LX)>$

Expressions (1) and (2) are tuple expressions describing the flow-state for IPv6 and ILNPv6 respectively. In expression (1), we can see that the transport protocol (TCP), the network layer (IP) and link layer (if) are using the IP address (A) to maintain the state of the connection. This will cause issues when the IP changes due to the mobile node moving across the network. In expression (2), we can see that ILNP uses the identifier (*NID*) at the transport layer (TCP) and uses the locator (*L64*) at the network (IP) and link (if) layer [3]. Instead of deploying a new server, ILNP uses DNS to store identifier and locator mappings [20]. End-systems can look up the Identifier-Locator Vector (ILV) for another end-system. This will simply be a list of *NID* and *L64* tuples. The use of locators allows the mobile node to join multiple IP networks simultaneously by binding its *NID* to multiple *L64*. This in turn allows the node to perform a "make before break" hand off, essentially connecting to the new network before breaking from the old one[19]. ILNP has proven to have "zero gratuitous loss during hand off, and very little misordering, as well as very consistent packet transfer and throughput" [3] compared to Mobile IPv6. The main issue with the ILNP solution is, although we don't need to update or deploy new middleware, we still need to modify the operating system's kernel. The kernel needs to use DNS to look up ILVs. The kernel also needs to be able to distinguish ILNPv6 from IPv6 packets. Furthermore, the kernel needs to keep track of all the active ILNPv6 connections in a map. Finally, we need to make changes to the kernel to manage hand-overs between these connections[19]. The aim of this project is to remove this overhead with the use of our application.

5 Requirements Specification

5.1 User Requirements

- R1: As a user, I must be able to configure my node's state, so I can choose to set my node to a router or a mobile state on the emulated network (O3, O4).
- R2: As a user, I should be able to configure my node's networks, so that I can choose which emulated networks my nodes are connected to and be able to create a custom network topology (O3).
- R3: As a user, I should be able to define my node's NID and FQDN (O1, O4).
- R4: As a user, I should be able to use the API to send a message from one mobile node to another node on the same overlay network without the need for a router (O4).
- R5: As a user, I should be able to use the API to send a message from one mobile node to another node on a different overlay network using a router node (O4).
- R6: As a user, I should be able to send a message by either specifying the destination NID or destination FQDN (O1, O4).
- R7: As a user, I should be able to move my nodes across the network (O5)

5.2 Functional Requirements

- R8: A node should be able to communicate with all other nodes as long as a router connects their corresponding overlay networks (O1, O3).
- R9: A node should implement an address resolution protocol for when a node wants to know if another node is present on their emulated network, they should send out a Neighbour Solicitation using the destination NID (O3).
- R10: A node should be able to receive Neighbour Solicitations and send out Neighbour Advertisement (03).
- R11: A node should be able to receive Neighbour Advertisements and keep an up-to-date map of NIDs to Locator, to be able to send packets using NIDs (O1, O3).
- R12: A node should be able to look up a node's Locator and NID in the fake DNS for name resolution. The node will send out a DNS query on a DNS multicast that all nodes are connected to. The DNS query will contain the wanted node's FQDN (O1, O2, O3).
- R13: A node should be able to receive DNS queries from the DNS multicast and answer with DNS responses. This should contain a time-to-live (TTL), one to many locators (L64), an identifier (NID) and the requested FQDN (O1, O3).
- R14: A node should be able to receive DNS responses and store the result in a map, so users can send packets using FQDN (O1, O3).
- R15: A node should be able to move across overlay networks without breaking the end-to-end principle (O5).
- R16: A node should be able to perform backwards learning to discover locators on the network. A node should do this using RREQ as part of the reactive ad-hoc protocol (O3).
- R17: A router should be a node, but it can also forward packets (O3).
- R18: A router should be able to receive router requests and perform backwards learning itself to forward the request across the network (O3).
- R19: A router should respond to router requests with router responses once it knows where the locator is on the network. The router response should also include the hop count which is the least amount of routers between the current router and the desired locator (O3).
- R20: The router should be able to forward packets between the multicast networks using the Locator (L64) part of the IPv6 destination address at the ILNP layer (O1, O3).

5.3 Non-Functional Requirements

- R21: The nodes will be implemented in Rust, deployed into individual Podman containers (O1).
- R22: The nodes must be able to deploy on the lab machines (O1).
- R23: The emulated network will be an overlay network, emulating ILNP over IPv6 using multicast (O2).

6 Software Engineering Process

The first two weeks were reserved to do the necessary reading and meet with my supervisor to strengthen my understanding of the problem. From this, I could write out requirement specifications. We decided to use the Agile Methodology approach. I would meet once a week with my supervisor. This would be a scrum meeting lasting between 1h and 1h30. Each week, I would design diagrams and code to cover the tasks for that week. At the end of the week, I could present my diagrams to explain my design decisions and use a prototype to record demos of the tasks completed. I would write an Agenda covering the progress for that week as well as the work I intend to do the week after. The Agenda also included questions I had that was stopping me from moving forward. I would send the Agenda two days ahead of my meeting with my supervisor.

7 Ethics

This project does not present any major ethical concerns. No external data is being used in this project. Ethical approval was given by the University of St-Andrews.

8 Design

In this section, we will design the protocol from the ground up. Figure 5 is an example network topology to explain what we'll be designing. The first two objectives is to build an ILNP emulation on top of IPv6 and UDP and emulate different networks using multicast groups. The L locators in Figure 5 represent the different networks we are emulating using different multicast groups. *Node1* is connected to the first network and *node2* is connected to the sixth network. *Router1* is connected to the first and second network and can forward packets between the two networks. Similarly, the other routers provide connection with the other networks. Objective 3 is to allow users to set up custom networks with custom traffic flows. As a result, nodes and routers need to be able to discover each other's identity and location. We use address resolution, name resolution (fake DNS) and path discovery. Domain Name System (DNS) is typically used for name resolution. DNS makes use of DNS servers to resolve FQDNs. It uses a combination of UDP, TCP, multicast and unicast [21]. For simplicity, the protocol will use a *fake* DNS which is a multicast channel reserved for DNS messages. Nodes will respond directly to DNS requests if it concerns them instead of the use of a third-party server. Because we are dealing with mobile nodes that can leave and enter the network at any moment (objective 5), we will be implementing a simplified version of the Ad-hoc On Demand Distance Vector Routing (AODV) protocol [22] for path discovery. This will allow routers to be unreliable mobile nodes and new paths can be discovered dynamically in a decentralised network. Address resolution will be used to discover neighbouring nodes and routers (nodes in the same multicast group).

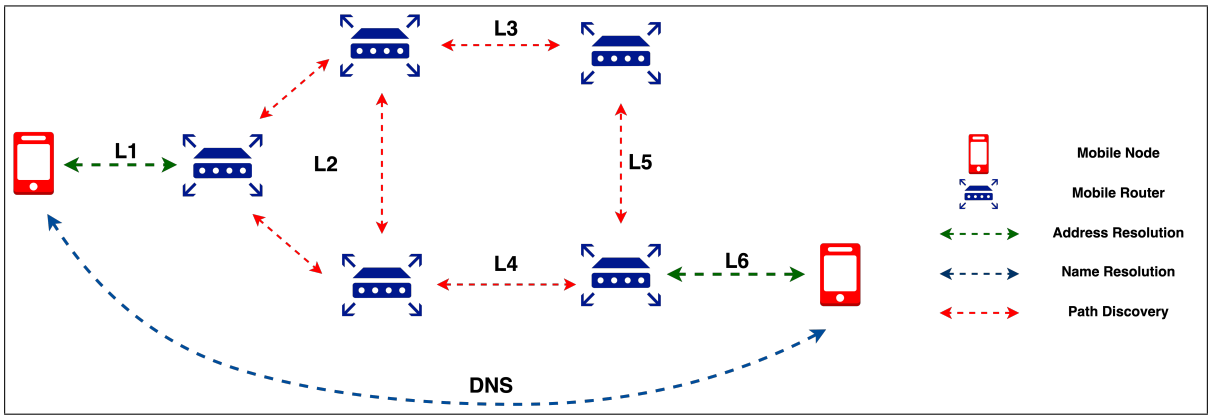


Figure 5: 2 nodes, 5 routers topology to demonstrate the end goal

The design section attempts to describe the implementation of the protocol from different perspectives. First is a description of how the networks are set up. Next, we look at layer diagrams to see how the payload is encapsulated before being released onto the physical layer. Thirdly, the application flow charts explain each layer's responsibilities. The state diagrams demonstrate the actions and triggers of our protocols. Finally, the sequence diagram shows how nodes interact using the protocols.

8.1 Network Set Up

In this section we briefly discuss how we set up the emulated networks using IPv6 multicast groups to cover objective 2. Only nodes connected to the same multicast groups can exchange packets between each other. Otherwise, packets will need to be forwarded using routers.

8.1.1 User ID

Each user on the Linux laboratory machines is assigned a unique 16-bit decimal as an ID (obtained via *id -u*). Because it is a 16-bit decimal, students utilise this ID as their application port number during development to prevent interference between different user's applications. We will make use of this ID to ensure multiple users can run the protocol independently without affecting each other's instances.

8.1.2 Multicast Underlay Network

Using the example in figure 5, *router5* is connected to the fourth, fifth and sixth network. The config file, for that node, is seen in figure 6. When setting up the multicast networks, the first 16 bits of the multicast IPv6 address are "`0xff02`" for link-local multicast addresses. The next two 16-bit blocks are zeroed. The user's unique ID is then used to complete the first 64 bits of the multicast IPv6 address (`ff02:0:0:5d73` in my case). This ensures each user has their own networks when running this protocol simultaneously. The network number from the config file is then converted to a 16-bit decimal and becomes the last 16 bits of the multicast IPv6 address. For the DNS address, we use the last 32 bits to not collide with the other multicast groups that only use the last 16 bits. Finally, we also make use of a multicast group to send all logs, errors and metrics, so we can run performance tests. This yields the following interfaces:

```
[node]
router = true
networks = [4, 5, 6]
nid = 0x0000000000005555
name = "router5"
```

Figure 6: *Router5*'s configuration file (Config5.toml)

- *multi1* : `ff02:0:0:5d73::4`
- *multi2* : `ff02:0:0:5d73::5`
- *multi3* : `ff02:0:0:5d73::6`
- *dns* : `ff02:0:0:5d73::5353:5353`
- *logs* : `ff02:0:0:5d73::5d73:5d73`

By completing the configuration file in Figure 6, the user is able to choose if the node is a router or not. This covers the first user requirement (R1). Secondly the user is able to decide which simulated network the node will be connected to which will in turn allow them to set up a custom topology (R2). The user is also able to define the node's FQDN and *NID* (R3).

8.1.3 ILNP Overlay Network

Multicast and unicast packets are encapsulated with an additional IPv6 header that will use the ILNP protocol. Using the example in figure 5, *node1* (*nid*: `0x::6666`) is sending a message to *node2* (*nid*: `0x::7777`). NID and Locators are used at the ILNP layer. In this example node one, on network one, is sending a message to node two, on network six:

- *src* : `0:0:0:1::6666`
- *dst* : `0:0:0:6::7777`

The last 16-bits of the multicast IPv6 address in the underlay network becomes the last 16-bits of the Locator at the ILNP layer. This is so we can identify the network and route packets through it using ILNP. This means the user is restricted to 65536 simulated networks. The last 64 bits is the node's identifier. This allows us to identify different nodes in the network, so we can forward packets to the right recipient. Because it is 64bits, the protocol can in theory handle a large amount of nodes. This proves we are using the ILNP addressing architecture over IPv6 multicast groups which conforms with the non-functional requirement R23.

8.2 Network Layers

In this section, we demonstrate how the user's data is encapsulated before being sent to the physical layer. The encapsulation allows us to set up nodes and routers to emulate ILNP over IPv6 (R23). Note that ILNP does not require additional packet encapsulation if the end-system kernels are updated with changes mentioned in RFC6743[17] and RFC6744[18]. However, the project requirement is to avoid modifying the kernel and to build ILNP on top of IPv6. This is also our first and second objective. Because of this, encapsulation is required to separate IPv6 and ILNPv6 headers. This will introduce additional overhead.

8.2.1 Layers Overview

Figure 7 is an overview of the user's payload as it travels through the layers before being released to the physical layer. A JTP header is applied to the payload for the JTP protocol. The ILNP header is then applied to account for the ILNP protocol. Emulated nodes and routers will make use of this header to read or forward packets on the simulated network. UDP and IPv6 headers are then also applied. In this project we will be implementing the orange part of Figure 7. This will encapsulate the user's payload with the JTP transport protocol header using the JTP API. We will then encapsulate the payload with an ILNP header using the ILNP API. We will finally use system APIs to transmit the resulting payload using UDP and IPv6. The system will encapsulate our payload with UDP and IPv6 headers and use the protocols as described in [23] and [24] respectively. Any nodes connected to the same multicast IPv6 address will be able to capture the UDP packets. The application receives a payload containing an ILNP header, JTP header and the user's payload. The application can then emulate ILNP behavior using information stored in the ILNP header. The user's payload is then released to the user using a JTP API. This will help with objective 1 and 4.

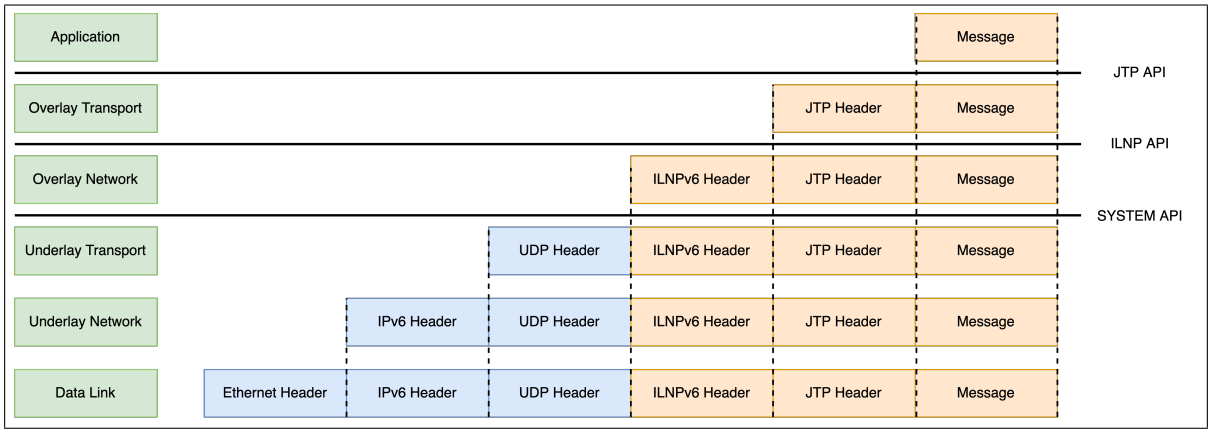


Figure 7: Final network stack above the physical layer and available APIs

8.2.2 IPv6 and UDP

Figure 8 is a recap of the IPv6 header and represents the network layer of our underlay network, i.e. multicast and unicast [24]. This covers our second objective. Figure 9 is a UDP header and is the transport layer of our underlay network [23]. For simplicity, the ILNP, JCMP and JTP protocols make use of the length field in the UDP header to handle the user's payload by subtracting their own headers from the UDP payload. This means we won't need a length field when designing our packets.

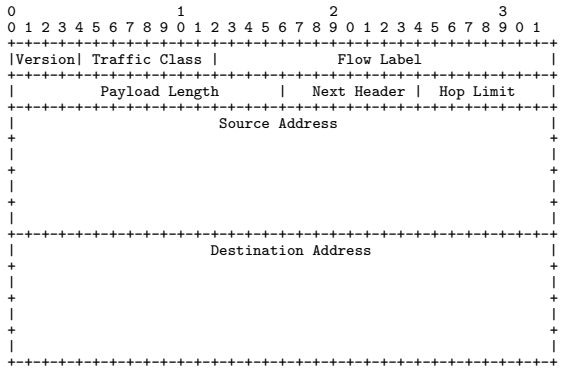


Figure 8: IPv6 header

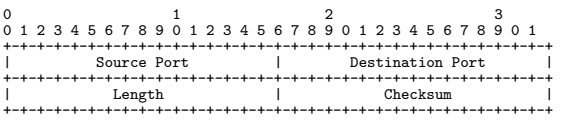


Figure 9: UDP header

8.2.3 ILNP Link Layer

Figure 10 represents an ILNPv6 header and is the network layer of our overlay network. This covers our first objective. In the Next Header field, we decided to use 150 for JCMP packets and 151 for JTP packets as they are unassigned and free to use according to the Internet Assigned Numbers Authority [25]. The identifiers and locators in the header are used to reconstruct the response header. The source locator is also used to determine which network (interface) a packet came from. These fields are also used as part of the DNS protocol, as discussed later in this paper. We do not make use of the *traffic flow*, *flow label* and *hop limit* fields, but we have included them for consistency and to be used when expanding this project.

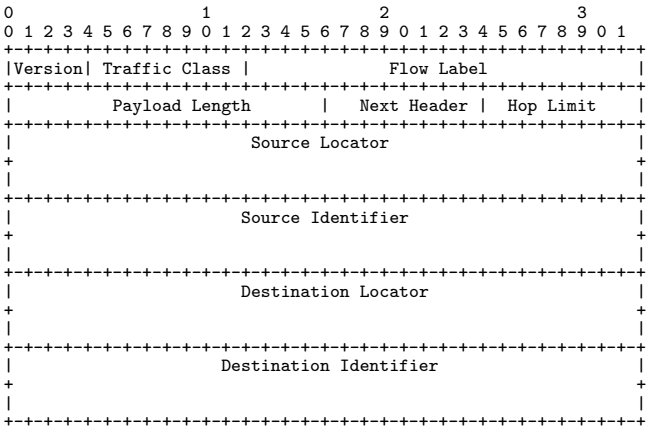


Figure 10: ILNPv6 Header

8.2.4 JCMP - Control Message Protocol

To conform to objective 3, we need to allow the user to create custom networks using custom nodes connected to specified multicast groups. For these nodes to be able to exchange data or forward data they need to be able to locate and identify themselves using ILNP (objective 1). For this we need a signalling protocol. We originally decided to separate signaling from the transport protocol to avoid the transport protocol's overhead. However, it became necessary when we decided to perform discovery in the multicast groups and send user data directly using local IPv6 addresses and ephemeral ports through unicast. User data sent in multicast groups would storm the network and introducing potential security concerns. Figure 11 is an illustration of using JCMP in the multicast groups and JTP for unicast transmissions. Figure 12 represents a JCMP packet and is the Control Message Protocol built on top of ILNP to convey messages between the nodes through the multicast groups. It is a simplified version of the Internet Control Message Protocol (ICMPv6) [26]. It is extended to manage address resolution, name resolution and path discovery. We decided to include it all in the same protocol for simplicity. Figure 12 is the required header for all JCMP packets.

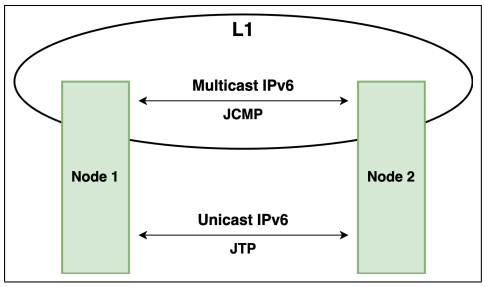


Figure 11: Multicast vs Unicast Packets

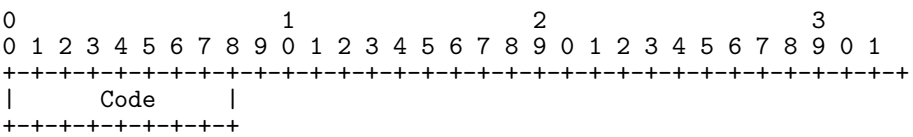


Figure 12: JCMP Header

8.2.5 JCMP for Address Resolution

The network layer is built on top of the link (data link) layer [7]. When Ethernet is used at the link layer and IPv4 at the network layer, address resolution is performed using the Address Resolution Protocol (ARP) [27]. This protocol resolves the destination Mac Address for a given IPv4 address, so the packet can be sent by Ethernet. IPv6 has its own enhanced implementation called the Neighbour Discovery Protocol (NDP) [28]. Because our ILNP protocol is built on top of IPv6 and not Ethernet directly, JCMP will implement its own simplified Neighbour Discovery Protocol that will resolve an IPv6 address and port number for a given identifier (NID). If the node associated with that identifier is present on the simulated network it will respond with its IPv6 address and unicast port number. Neighbour discovery is done in the multicast group, so all nodes in that network can respond. However, once we've resolved

the destination IPv6 address and port number, we can forward the packet directly using unicast. This is demonstrated in Figure 11.

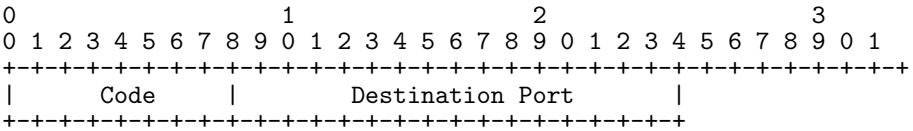


Figure 13: JCMP Header for Neighbour Advertisement (NA) in address resolution

Code *0* is used for Neighbour Solicitations (NS) using the JCMP header in Figure 12, at the ILNP layer the source and destination locator becomes the locator of the network we are performing a discovery in. So if a node is multi-homed, the node will send two separate NS on each network and append the appropriate locators in the ILNP header. The source identifier is the node's NID and the destination identifier is the NID of the device we are looking for. Code *1* is for Neighbour Advertisements (NA). Nodes on the same multicast group can listen for NS and respond if the destination identifier at the ILNP layer is equal to their identifier. If the identifier matches they can respond with the JCMP header shown in Figure 13. Upon receiving a NA, the source IPv6 address is recovered from the IPv6 layer and the source port is recovered from the JCMP header. Code *2* and *3* were used for Router Solicitations and Advertisements. This became redundant after implementing path discovery which we discuss later in this paper. Reusing the IPv6 source address at the IPv6 layer and the *NID*s and Locators from the ILNP layer allowed us to reduce packet size and so bandwidth usage. It also allowed us to keep the code simple, reducing chances for error. After performing address resolution, the sending node has the IPv6 address and port number of the receiving node. Using the IPv6 and port number we can send data packets directly using unicast in the underlay network. This saved us from storming the network by broadcasting data packets in the multicast groups. This will help implement R9, R10 and R11.

8.2.6 JCMP for Name Resolution

JCMP is also extended to handle DNS messages. Figure 14 is a DNS FQDN Query and uses the JCMP header with a FQDN as the payload and using code *4* as the packet code. This packet is used to request the Identifier-Locator Vector (ILV) associated with the given FQDN. We don't need to add a length field for the variable FQDN field as a length field is included in UDP in the transport layer of the underlay network. At the ILNP layer, the source locator, destination locator and destination identifier takes a placeholder value of *::53535353*. The source identifier is the sending node's identifier (NID).

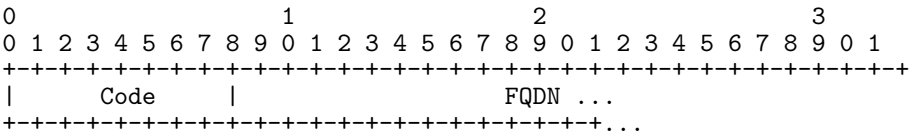


Figure 14: JCMP DNS FQDN Query header

All nodes are listening on the DNS multicast groups no matter which simulated network they are connected to. If the node's FQDN is equal to the requested FQDN (given as *name* in the config file, see Figure 6) the node will respond with a DNS FQDN Response. This is shown in Figure 15.

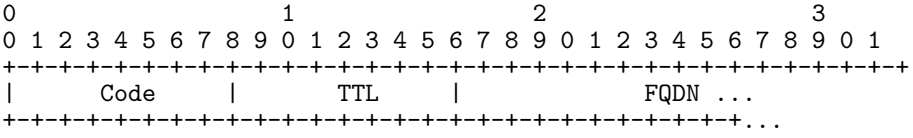


Figure 15: JCMP DNS FQDN Response header

This again uses the JCMP header but is extended with a Time-to-Live (TTL) and FQDN field. The TTL is used to say how long the name to ILV mapping is valid for. The FQDN is added again in the response so that the sending node can distinguish between different DNS responses. For simplicity, the node generates a DNS FQDN Response for each ILV entry. So if the node is multi-homed it will generate a

response for each Locator it is connected to. Each ILV entry is inserted in the source locator and identifier field at the ILNP layer. The destination identifier is recovered from the DNS FQDN request packet and the destination locator takes the *::53535353* placeholder. The node requesting ILV using a FQDN can recover the Locators and NID from the DNS FQDN responses at the ILNP layer. Global DNS uses third-party servers and cannot override source address with target address in their DNS responses. And so the DNS answer is included in the packet's payload [21]. However, because we are using a simplified DNS implementation where we query the nodes directly, we can use this technique to reduce packet size. The packet code for a DNS FQDN response is code *5*. This will help implement R12, R13 and R14.

8.2.7 JCMP for ILV Resolution

As an additional feature, we decided to allow users to send messages using the identifier (NID) instead of FQDN. For this, we extended the fake DNS to respond to NID lookups. The DNS ILV Query simply uses the JCMP header with packet code *6*. The target NID is set in the destination identifier field at the ILNP layer. The response is a DNS ILV Response and is represented in Figure 16. The packet code here is code *7* and the TTL is used to say how long the ILV mapping is valid for. Again, the requested identifier is returned along with the locator in the source fields in the ILNP header.

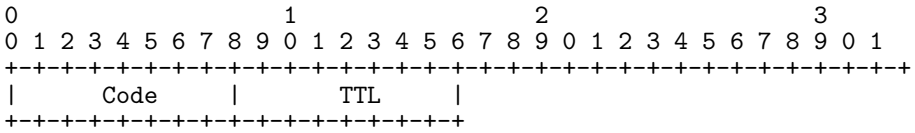


Figure 16: JCMP DNS ILV Response header

8.2.8 JCMP for Path Discovery

JCMP is also extended to help nodes discover paths along the network using simplified versions of Ad-Hoc On-Demand routing requests (RREQ) and routing responses (RRES) [22]. Figure 17 represents a JCMP Router Request. It is used to discover a path for a particular locator. It uses the packet code *8*. When a node is multi-homed, a request packet is generated for each interface to discover a path through each simulated network the node is connected to. At the ILNP layer, the source and destination locator are the source locator we are performing path discovery in. The destination locator is the current locator so routers can know where the request came from and not perform discovery on that path. The source identifier is simply the sending node's identifier. The destination identifier uses a placeholder value of *::ff02ff02* as the request is open to all routers on the network. The JCMP header is extended in Figure 17 to include a Destination Locator. This is the target locator, the locator we are trying to discover a path to. When a router receives a router request it will need to perform path discover themselves to be able to respond. The router will recover the *Hop Count* field, and extract one from it before using it in its own router request packets. If the *Hop Count* is equal to 0 the router will give up trying to discover the path. This is to avoid infinite looping [29]. We decided to set the maximum hop count to $2 \times nb\_nodes + 1$ to account for packet drops.

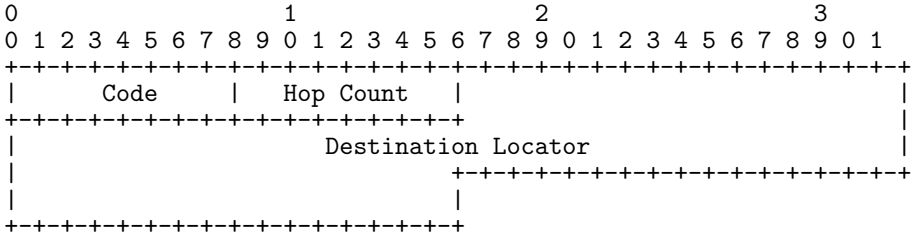


Figure 17: JCMP Router Request (RREQ) header

In response, other routers can respond with a JCMP Router Response shown in Figure 18. It is used to respond to a path request for a particular Locator. Packet code *9* is used here. As the response travels through the network, *Hop Count* is incremented by one. When a router receives a routing response packet, they know they can access the target locator through the router that sent the response. It knows

how many routers the packet has to travel through to reach the requested network thanks to the *Hop Count* field. It can then prioritise which path to use when sending data packets based on the *Hop Count*. The response also includes a Time-to-Live (TTL) which indicates how long the path is valid for. This value is decided by the router residing in the destination network (retrieved from its config file). This will help implement R16, R18 and R19.

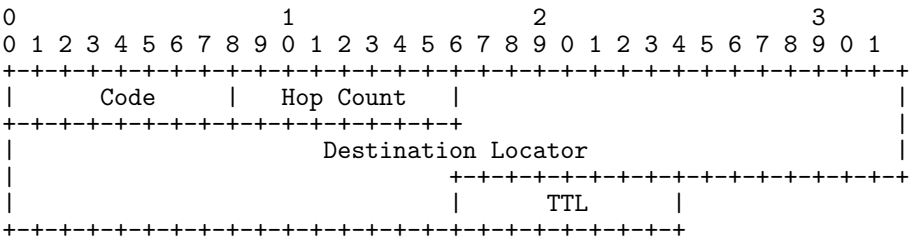


Figure 18: JCMP Router Response (RRES) header

8.2.9 JTP - Transport Protocol

Figure 19 represents a JTP packet and is the transport protocol of our overlay ILNP network. We didn't need a length field here as we can use the one in the UDP header of our underlay network. No header was required here. User data sent from nodes are sent using JTP encapsulated by an ILNPv6 header. This will help implement R8. The header can be extended in the future to make the transport protocol more reliable.

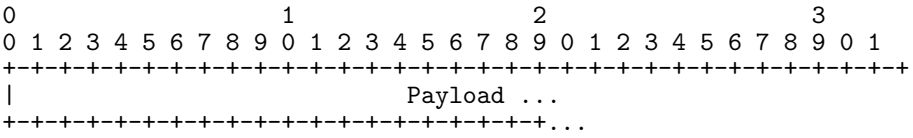


Figure 19: JTP header

8.3 Application Flow Chart

In this section, we look at UML flow chart diagrams. These diagrams will explain the different responsibilities each layer will take on. JTP is responsible for sending and receiving user data. The ILNP layer is responsible for discovering nodes and forwarding user data packets accordingly. The system layers is responsible with handling sockets and sending packets through the actual underlayer network.

8.3.1 Open Socket

Before a user can use the JTP protocol, they need to open a JTP socket using the JTP APIs. On opening a socket, the command is initially sent to the underlay layer. Here we open and configure a socket and bind it to the Ethernet interface. We then join the IPv6 multicast groups specified by the config files as well as the DNS and Log multicast group. We then release the socket to the INLP layer. This layer set ups three asynchronous thread to receive and handle packets from the UDP sockets. JCMP packets are received and handled in the first thread. The second thread is for consuming JTP packets into a queue while the third thread is for processing JTP packets from the same queue. We were experiencing packet drops when sending large amounts of data packets. Consuming the data packets into a separate queue as quick as possible greatly reduces packet loss. JTP packets are released to the user. This is shown in Figure 20.

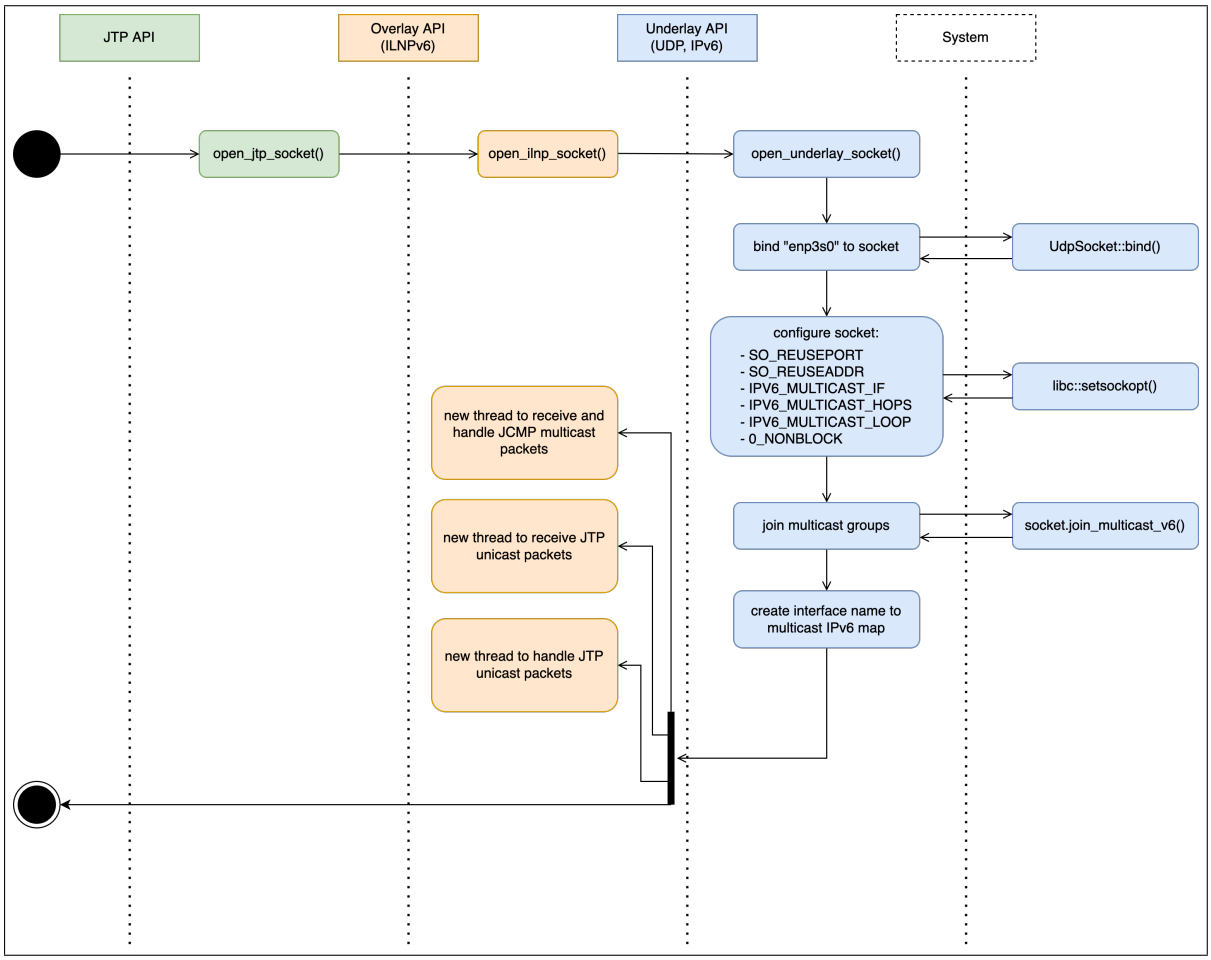


Figure 20: UML flow chart for when the user opens a socket

8.3.2 Address Resolution

Address Resolution is required to send packets to nodes on the same simulated network as the sender. It is also required to be able to forward packets to a router if the receiver doesn't reside in the same network. When invoking the address resolution function, the ILNP layer will first try to find the IPv6 and port number associated with the given *NID* in the table. This can be the *NID* of a neighbouring node or router. If it cannot find it, it will then transmit a JCMP Neighbour Solicitation (see Section 8.2.5) and wait for *ND_RTO* (set in the config file) microseconds as a transmission timeout. This process is repeated until *ND_RETRANSMISSION_LIMIT* (set in the config file) is reached and the Address Resolution has failed. If however, after sending a JCMP Neighbour Solicitation, the JCMP receiver thread (mentioned in Section 8.3.1) receives a JCMP Neighbour Advertisement from a neighbouring node, Address Resolution will succeed and we can then send a packet using unicast at the underlay network.

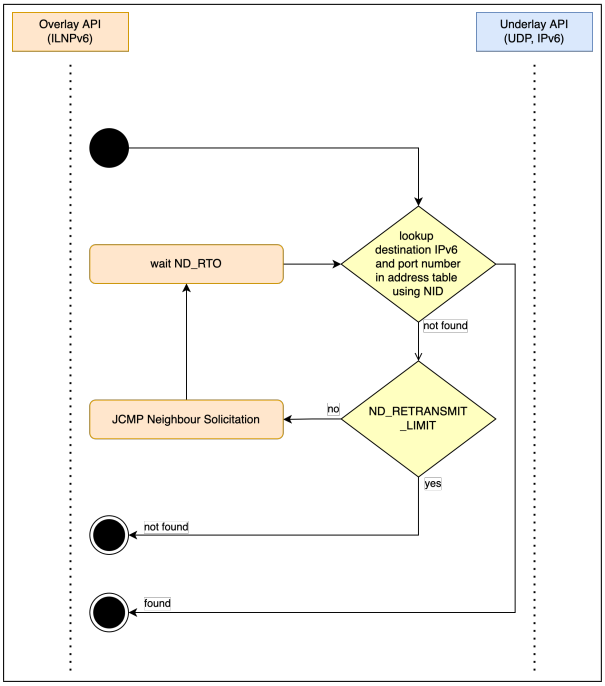


Figure 21: UML flow chart for address resolution at the ILNP layer

8.3.3 Name Resolution

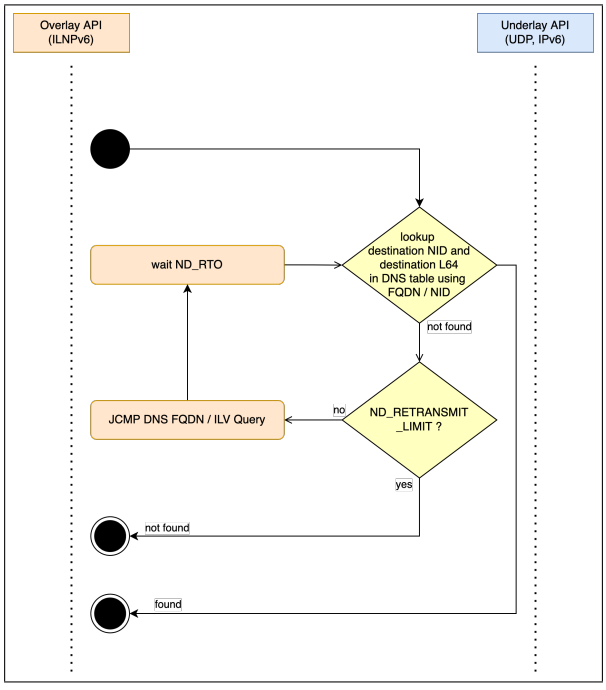


Figure 22: UML flow chart for name resolution at the ILNP layer

Name Resolution is required to determine the ILV for a receiving node so that we can forward a packet using Locators in ILNP. Standard DNS will resolve a FQDN for a static IPv6 address [21]. For ILNP, DNS is extended to handle multiple locators for a given identifier (NID) to allow multi-homing, and gracefully switching between networks [20]. Figure 22 is a function used at the ILNP layer to resolve ILV in our fake DNS. First the function will attempt to retrieve the ILV associated to a given FQDN or NID in the DNS tables. If this fails, it will send a JCMP DNS FQDN Query to resolve using FQDN or it will send a JCMP DNS ILV Query to resolve using a NID. The function will then wait for *ND_RTO* microseconds until looking up the tables again. This is done *ND_RETRANSMISSION_LIMIT* times and then name resolution fails. If the JCMP receiver receives a JCMP DNS Response, name resolution will succeed and ILV (array of Locators/NID) is returned to be used for forwarding packets.

8.3.4 Path Discovery

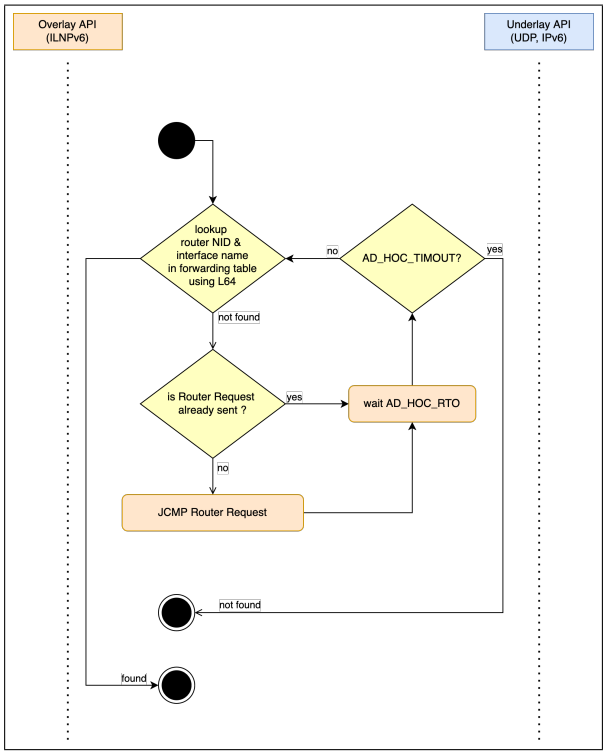


Figure 23: UML flow chart for path discovery at the ILNP layer

Path Discovery is required by all nodes and routers when Address Resolution fails. If Address Resolution fails but DNS lookup succeeds, it must mean the node resides on a separate network. In this case we will try to resolve the path to this network. Firstly, the function will attempt to retrieve a router NID and interface name for a given target locator in the forwarding table (note that the interface name is simply the name associated to the locator we are connected to where the router resides). If multiple entries are returned, the function will return the router NID with the least number of hop counts. The router NID, is the identifier of the router we need to forward the packet to for the packet to reach its destination. The hop count is the number of routers the packet will have to travel through before arriving to its destination. If no entries are returned from the forwarding table, the function will attempt to discover the path. A JCMP Router Request is sent. The function will then keep checking the forwarding table for *AD_HOC_TIMEOUT* milliseconds. It is waiting for the JCMP receiver to receive a JCMP Router Response. The wait will be proportional to the number of routers between the sender and the receiver as all routers on the path will need to perform path discovery also. We decided to add a *AD_HOC_RTO* in nanoseconds to avoid exhausting computational resources.

8.3.5 Send Message using NID

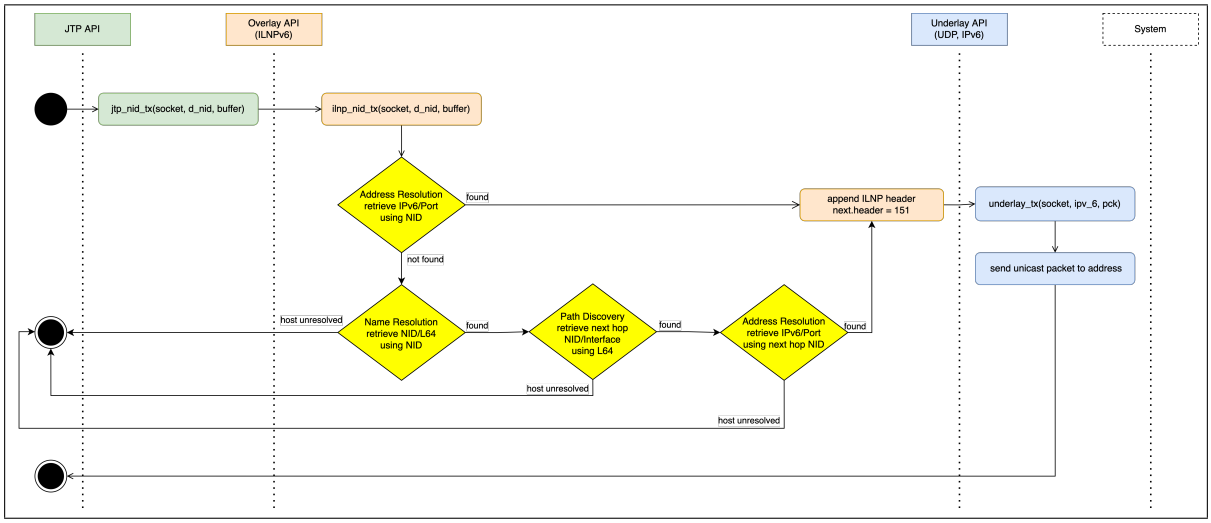


Figure 24: UML flow chart for when the user transmits data using NID

The flow chart in Figure 24 demonstrates how the user uses the JTP protocol to send messages using the destination identifier (NID). The payload goes straight to the INLP layer. The ILNP layer first attempts address resolution to check if the receiving node is on the same network as the sending node. If this succeeds we can wrap the payload with an ILNP header and send it. If this fails, the next step is to

perform name resolution to check the node is active and determine which networks the node is connected to. If name resolution succeeds we need to discover the path to the Locator associated with the identifier (NID). Next we need to resolve the address and port for the router we are forwarding the packet to. Finally, we can encapsulate the payload with the appropriate ILNP header and send the packet. This covers R4, R5 and R6.

8.3.6 Send Message using FQDN

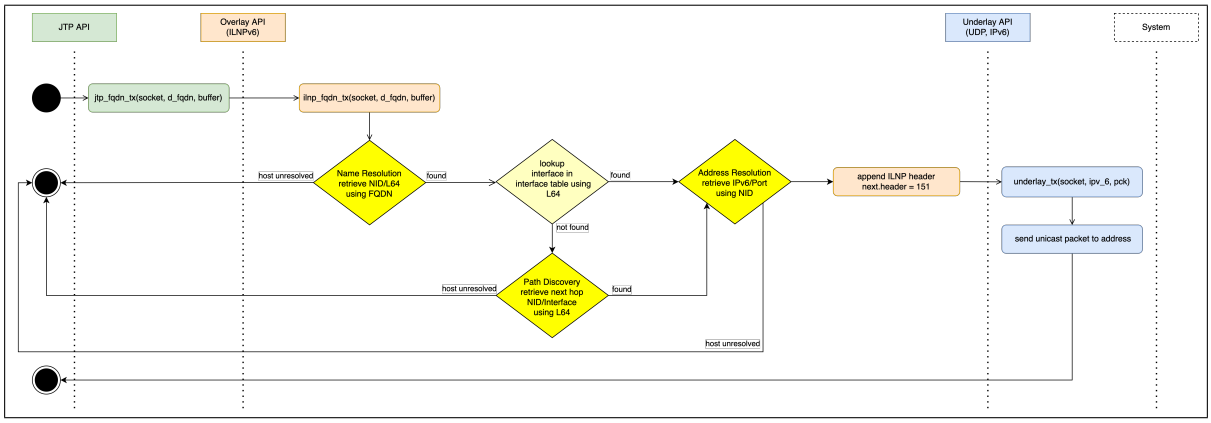


Figure 25: UML flow chart for when the user transmits data using FQDN

The flow chart in Figure 25 demonstrates how the user uses the JTP protocol to send messages using the destination Fully-Qualified-Domain-Name (FQDN). The payload goes straight to the INLP layer. The ILNP layer attempts to retrieve the destination locator and the destination identifier (NID) for the given destination FQDN using name resolution. Next the ILNP layer will lookup all Locator from the ILV to determine if the sending node is connected to the same network as the receiving node. For each interface found, we perform address resolution to see if we can forward the packet directly. If this fails, it means the receiving node is not on our network and we need to perform path discovery. Once the next hop router is found, we perform address resolution to be able to forward the packet to the router. The packet is encapsulated with an ILNP header and sent. This also covers R4, R5 and R6.

8.3.7 Close Socket

Figure 26 is the flow chart for closing a socket. Here we go straight to the underlay layer to leave the multicast groups including the DNS and Log groups. We then drop the socket to close it and return to the user. In the ILNP layer, just before closing the socket, we send the Protocol Control Block (PCB) to the log multicast for processing. This is to be able to analyse how the node performed.

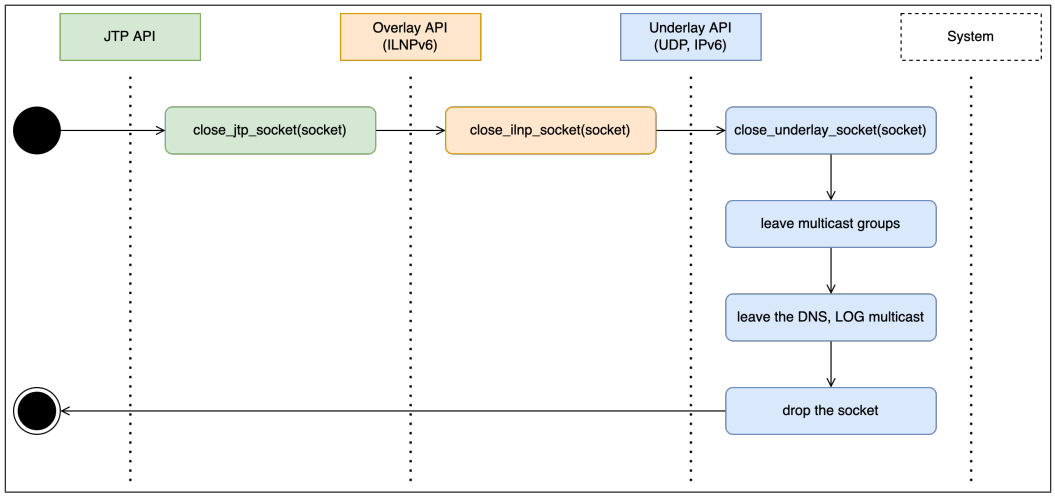


Figure 26: UML flow chart for when the user closes a socket

8.4 State Diagrams

The state diagrams will help us demonstrate how each protocol will react to incoming requests from both the network and the user.

8.4.1 JTP State Diagram

Figure 27 is a state diagram of our JTP protocol. The protocol can actively open to send packets either using the NID or FQDN. The protocol can also passively open to listen for incoming JTP packets and release them to the user. The user has the choice to either indefinitely wait for a packet, poll a packet or set a timeout to receive a packet using the *jtp_rx* function.

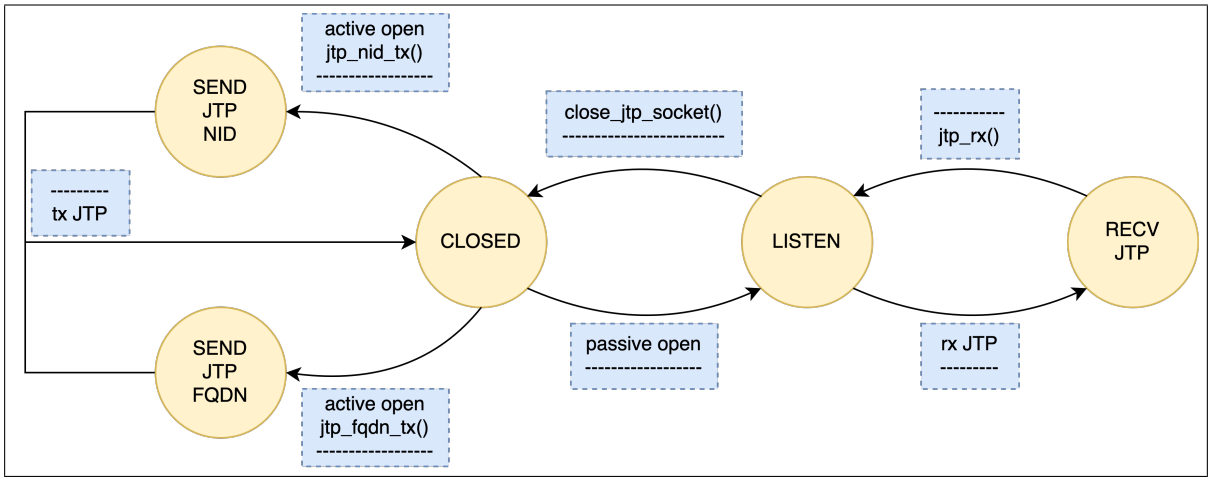


Figure 27: State Diagram for the JTP protocol

8.4.2 JCMP State Diagram

Figure 28 is a state diagram of our JCMP protocol. Using the API we can perform a passive open and stay idle listening for incoming JCMP packets. On receiving a packet, the protocol switches state and triggers a handler. The handler responds to neighbour solicitations with neighbour advertisements, responds to DNS queries with DNS responses and responds to route requests with route response packets. An active open can also be triggered by the user when sending a data packet. An active open is triggered and address resolution, name resolution or path discovery is performed and the appropriate requests are sent out to the network.

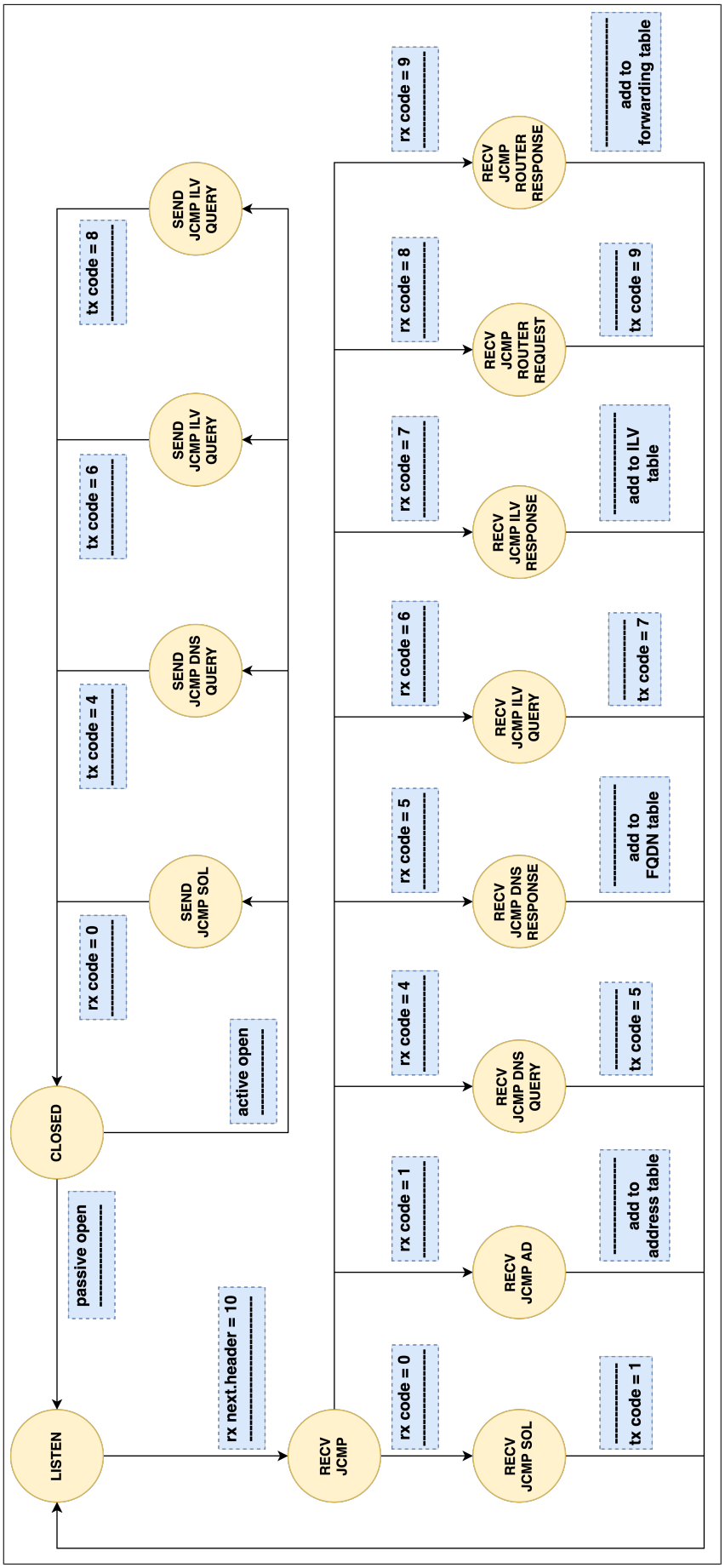


Figure 28: State Diagram for the JCMP protocol

8.5 Sequence Diagrams

In this section we will use Sequence Diagrams to explain how the nodes interact with each other using different protocols.

8.5.1 JCMP Address and Name Resolution

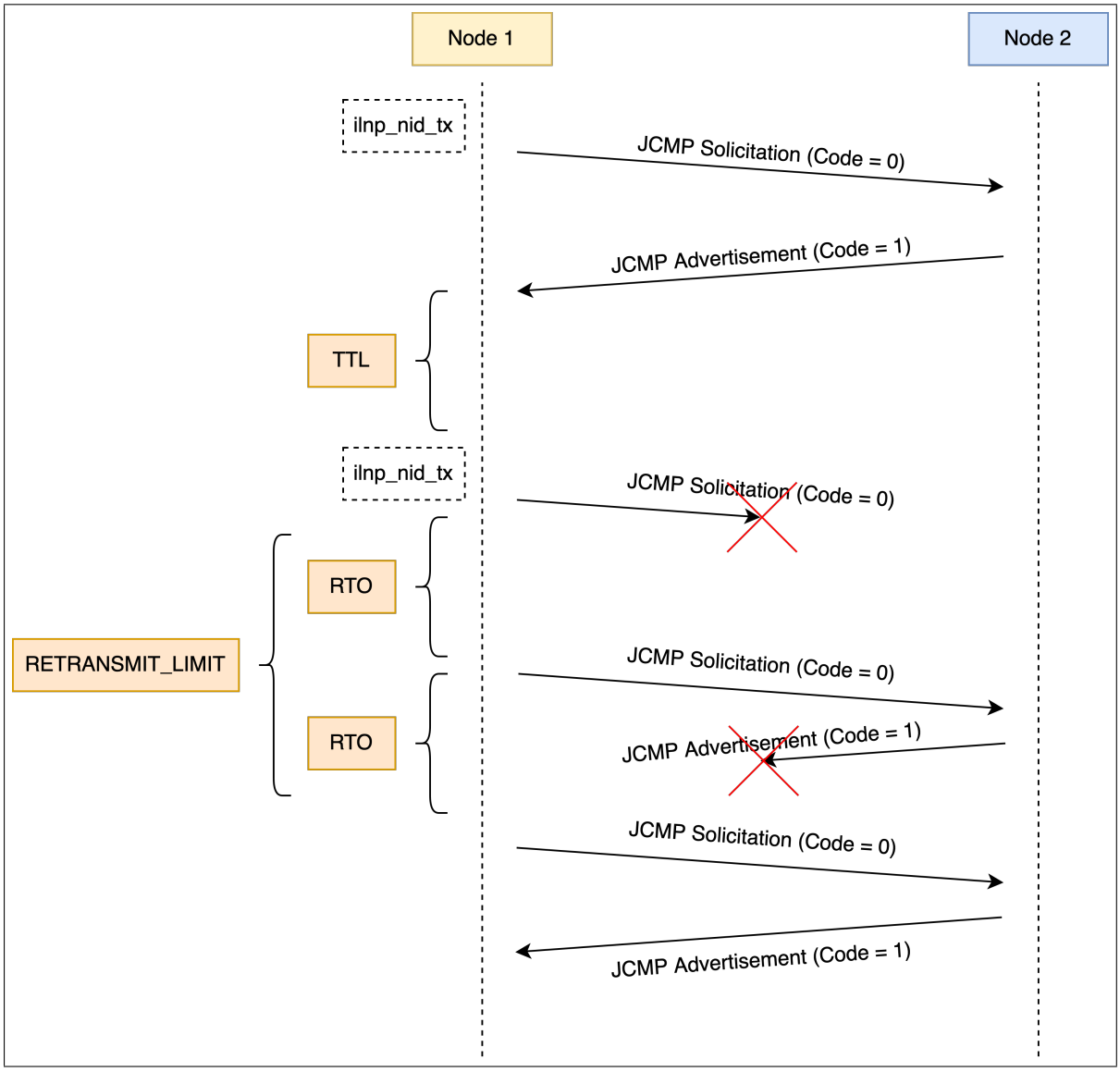


Figure 29: Sequence diagram for JCMP address resolution between 2 nodes

Figure 29 demonstrates address resolution but the same design is used for name resolution. The node sends neighbour solicitations until it receives a neighbour advertisement from the node the user wants to send a data packet to. This is attempted *ND_RETRANSMIT_LIMIT* times (configured in the node's config file) and then an error is returned to the user. *ND_RTO* (also configured in the config file) is the time the application will wait for a neighbour advertisement before sending another neighbour solicitation. *TTL* is the time the information received from the neighbour solicitation is valid for. After this we have to make a new request. Our fake DNS uses the same design. The difference is, DNS will use the DNS multicast and it will use JCMP FQDN Queries or JCMP ILV Queries instead of neighbour solicitations and JCMP FQDN Responses or JCMP ILV Responses instead of neighbour advertisement. The other difference is the *TTL*, which defines how long the DNS entry is valid for, is retrieved from the DNS response. This covers R9, R10, R11, R12, R13 and R14.

8.5.2 JCMP Path Discovery

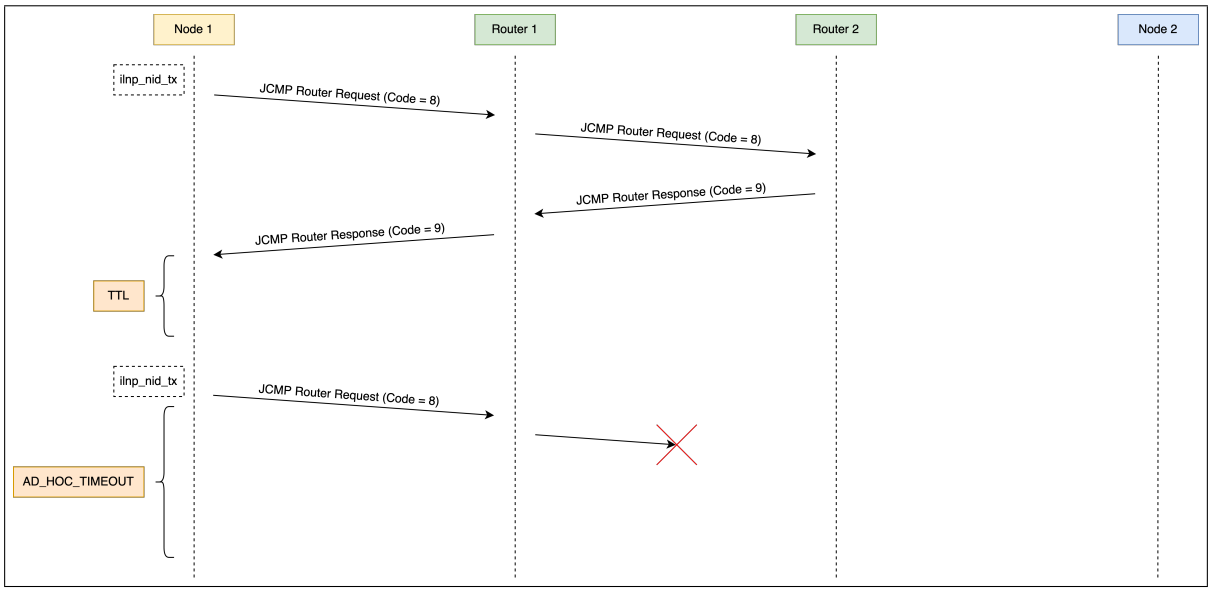


Figure 30: Sequence diagram for JCMP path discovery to a node across 2 routers

Figure 30 is a sequence diagram representing path discovery in the JCMP protocol. Node 1 sends out a JCMP Router Request containing the target locator. The target locator is *Node2*'s locator in this case. Each router along the path forwarded the requests until we reach a router residing in the target locator. This router knows it can access the requested network and so can already respond with a JCMP Router Response. Routers along the way can then also respond to pending route requests they have received. Eventually *node1* receives router responses from neighbouring routers with a *hop count*. *TTL* is the time the route is valid for until another discovery needs to be made. This value is retrieved from the JCMP Router Response. *AD_HOC_TIMEOUT* is how long the application will wait until it tells the user it failed to discover a path. We decided not to implement a retransmission as the retransmission timeout would depend on the number of routers in the network which would complicate the code. Instead, we decided to allow the application to discover other paths. Of course this fails if we use a topology like the one in Figure 33. If path discovery fails, no path will be discovered until another data packet is sent which will trigger a retransmission. The path discovery function is also able to accept a source interface. This happens when the router receives a route request from an interface and tells the path discovery function to not discover a path on that interface. This covers R8, R16, R17, R18, R19 and R20.

9 Implementation

9.1 File Structure

Figure 31 is the final file structure at the end of the implementation. All the rust code is inside the *src* directory. This covers the the non-functional requirement R21. The *layers* folder contains a folder for each layer of our protocol. *underlay_network* handles the logic involving the ethernet interface, UDP and IPv6. *overlay_network* contains the logic that handles the ILNP header built on top of IPv6. Finally, the *jtp_network* handles the JTP payload and provides an API for the user to send and receive data through the emulated network (objective 4). *deploy-node* is a bash script to deploy individual nodes. The *Dockerfile* is used to deploy the node in a podman container. The *deployment* folder contains scripts to *deploy* and *stop* a toplogy deployment which deploys multiple nodes simultaneously using the toplogy defined in the *topology.yaml* file. This simplifies the 3rd objective for the user. *Config.toml* files are generated in the *config* folder to be able to deploy different nodes with different configurations. The *hosts.txt* file stores FQDNs of the machines we wish to use when deploying the application to separate machines on the same local network. The *settings.toml* file contains network configurations such as *TTL*s and *RTO*s. This needs to be modified when running different performance tests and when running specific applications on top of the protocol. The *metrics* folder contains all the logs we collected when measuring the performance of our protocol. We use *process_logs.py* and *process_sensors.py* to generate plots from these logs. Finally, the *scripts* folder contains scripts to help debug multicast and unicast packets as well as help us find available and free end systems to use on the local network.

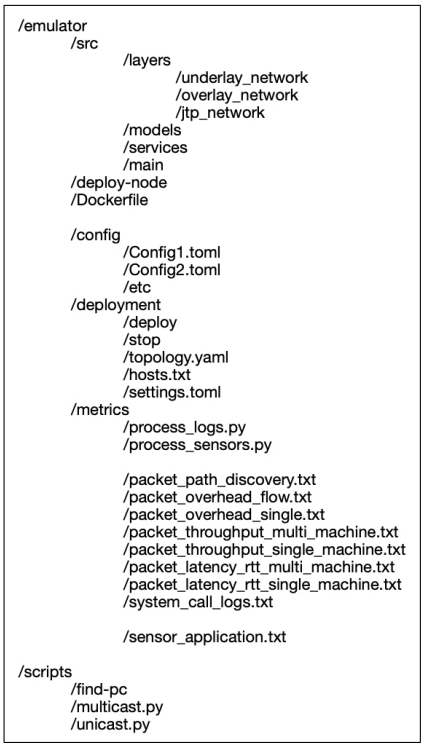


Figure 31: Code's file structure

9.2 Code Structure

Figure 32 is a representation of our main rust files. In the *main.rs* we have all the functions needed to run the performance tests. We also have a function to run the farming scenario (discussed further in the paper). There is a function for the logger to receive logs on the log multicast network and print them to the shell for collection. Finally the *main.rs* file has a *user_app* which allows the user to simply send messages to other nodes using the command line. This was simply to test the network as we were implementing it and to allow the user to try the protocol APIs before building their own application on top. The rest of the code can be extracted and used for the user's own use of the APIs. In the *underlay_network/mods.rs* file, we have *open_underlay_socket()* which is used to create sockets and join the node's multicast groups defined in the config file. We also have *close_underlay_socket()* for leaving the multicast groups and dropping the sockets. *INTERFACES* is a map that stores the emulated (multicast) networks the node is connected to. An interface name is mapped to a (Locator, Ipv6 multicast address) tuple. This way we can easily pass an interface name around to reference one of our emulated networks. *underlay_multi_tx()* is used to send a multicast packet using the interface name to identify which multicast Ipv6 address to send it to. *underlay_uni_tx()* is used to send a unicast packet using the destination node's local IPv6 address and ephemeral port number. For both these functions, it is assumed that the ILNP header is applied. *underlay_network* does not deal with anything ILNP related. The *underlay_network/under_socket.rs* file contains a function to create a multicast socket and a function to create a unicast socket. A lot of advanced configurations aren't available when using *std::net::UdpSocket*. Because of this, we decided to setup the sockets using *libc* in *unsafe* blocks [30]. We could then convert them to *std::net::UdpSocket* and then *tokio::net::UdpSocket* to allow the sockets to be shared across threads for asynchronous processing. This meant, we could have full control of the sockets and make the correct configurations before binding them to the interface. We tried using *socket2::Socket* but it did not behave as expected either. This covers objective 2.

Figure 32: Rust files for deploying a node in the emulated network

The *overlay_network/mods.rs* file is the main body for emulating the ILNP protocol. This is for objective 1 and 3. *CONFIG* stores the application's configurations (*TTL*s, *RTO*s, etc). It also stores which emulated network the node should connect to and so it is used by the overlay network too. Finally it tells the ILNP layer if the node is a router or not. Additionally it stores a number of boolean variables for the performance tests. This is because we needed to perform certain action depending on the test. For example, when measuring the time it took to perform path discovery, we needed to print the system's time before and after the discovery. This functionality is switched on and off using these booleans. *PCB* is the Protocol Control Block. In this variable we count the number of different packets the protocol is receiving and sending. This helped us diagnose the protocol during implementation. *NID_ADDRESS_RESOLUTION_TABLE* is similar to the ARP table. It stores a mapping of *NID* to

(interface name, IPv6 address, port number) tuples. This means from a given *NID* we can send a packet using unicast. *NAME_ILV_TABLE* is a name resolution table. It is used to store a mapping of *FQDN* to ILV. The key used in this map is (*NID*, *L64*) hashed. This is because, if the node is multi-homed, multiple locators will be returned for a given identifier. We decided to uniquely identify each entry with the identifier and locator combined. The assumption here is that all nodes are uniquely identified by their identifier and no 2 nodes can have the same identifier. Otherwise, we would be storing locators for the wrong nodes. *NID_ILV_TABLE* is another name resolution table. This table stores ILV from *NID* look-ups. Again, each (*NID*, *L64*) tuple is hashed to be used as the key. The *LOCATOR_FORWARDING_TABLE* is used to store the next hop. After performing path discovery, we receive the *NID* of the router, the target locator, the interface the we can access the router on and the hop count to reach the target locator. We store this in the forwarding table. Each entry is uniquely identifiable by the (router *NID*, target *L64*) tuple. This tuple is hashed and used as the key for the table. In this table, we originally also stored the router's IPv6 and port number and we would perform address resolution after path discovery. However, when performing path discovery, each router would perform address resolution each time they received a router response. This significantly reduce the performance of my path discovery. We didn't require the router's IPv6 and port number to respond to a router request, so we decided to remove it from the forwarding table. This simply meant we had to perform address resolution before forwarding a packet but this is required anyway. *ILNP_QUEUE* is a queue to store incoming unicast packets. This was necessary to free up the receiver and process JTP packets asynchronously. *open_ilnp_socket()* calls the the underlay API to open the sockets. It then creates a thread to receive and handle JCMP packets from the multicast socket, a thread to receive JTP packets and send them to the ILNP_QUEUE, and a thread to process the JTP packets. *close_ilnp_socket()* is used to send the PCB to the logger multicast group before closing the sockets using the underlay API. *ilnp_nid_tx* is used to send a packet using a destination *NID*. It is described in the design section. *ilnp_fqdn_tx* is used to send a packet using a domain name. This funciton is also explained in the design section.

The *overlay_network/overlay_handlers.rs* contains our handlers and discovery functions. The *handle_ilnp_multicast_buffer()* function is to handle the multicast packets received. This function will check if the source locator in the ILNP header is in the list of networks the node is supposed to be connected to. This was necessary when running the protocol on the same machines. We ran into the problem that if one node joins a multicast group, it forces the other nodes on the same machine to join that group as well. This is because all the nodes are bound to the same ethernet interface. This filtering is not required when running the nodes on separate machines. We also check if *Next.Header* is 150 for JCMP packets. The function then calls the *handle_jcmp_packet()* functions. This processes the jcmp request and responds accordingly. More information is available in the design section. To be able to run the nodes on the same machine, we needed to switch on *IPV6_MULTICAST_LOOP* in the multicast socket. This meant in the *handle_jcmp_packet()* we needed to ignore packets where the source identifier at the ILNP layer is equal to our identifier. If equal, it meant the packet received was a packet sent by the node itself. In this function, we also decided to accept advertisements which are not meant for us. We did this to reduce the packet overhead. *handle_ilnp_unicast_buffer()* is used to process the JTP packets. This simply checks the *Next.Header* is 151 for JTP packets. It then sends the data payload along with ILNP information such as source *NID*, *L64* and destination *NID*, *L64* to the *JTP_QUEUE*. *handle_router_forward()* is used to forward the packet when the JTP packet is not intended for us. *handle_destination_nid()*, *handle_destination_fqdn()*, *handle_destination_ilv()*, *handle_path_discovery()* are our address resolution, name resolution and path discovery functions. These are explained in the design section. The *overlay_network/jcmp_tx.rs* file contains all the functions to send JCMP requests.

The *jtp_network/mod.rs* file contains functions to use the ILNP APIs. It allows the user to open and close a socket. The user can also send data by either using a *NID* or *FQDN* to another node. The last API available is the receiver, *jtp_rx(timeout)*. This receives packets from the *JTP_QUEUE* and serves it to the user. The user can choose a timeout of -1 for blocking, 0 for pool or a positive integer. If a positive integer is used, the receiver will wait to receive a packet until the timeout expires (in milliseconds). We have created APIs for the user to use the protocol. This covers objective 4. The rest of the functions are helper functions. To create the packet structs in the *model* folder, we decided to use rust's modular bitfield library [31]. This ordered the bytes in big endian format. It also simplified creating packets with custom sized fields that were not a multiple of 8.

9.3 Running Emulator

To use the user application that uses the JTP APIs, the user needs to move into the *emulator* folder. The user can then run:

- ***./deploy-node NB*** to deploy a single node using a *ConfigNB.toml* file stored in the *config* folder where *NB* is a number used to number the config file. To kill the node, user simply needs to press *Ctrl+C* in the command shell.
- ***/deployment/deploy*** is used to deploy all nodes using the topology defined in *topology.yaml* and the settings defined in *settings.toml*. Config files will be generated automatically.
- ***/deployment/stop*** is used to stop all the nodes. This gives them a chance to close sockets and exit gracefully.
- ***/deployment/deploy -f hosts.txt*** is used to deploy the nodes on different machines using the FQDNs stored in the file. It's important hosts.txt contains the same number of nodes and routers in the topology plus one for the logger (e.g. if there are 2 nodes 3 routers, we require 6 hosts).
- ***/deployment/stop -f hosts.txt*** is used to terminate the nodes on all machines. It is important to include the file or the nodes will remain active.
- ***/deployment/deploy –help*** for help.

9.4 Running Performance Tests

To run the performance tests and the farming scenario discussed in the evaluation section. It is important here that the user modifies the *topology.yaml* and *settings.toml* to suit their needs. Furthermore, in the performance testing section, we increase the number of routers in our topology as we perform the test. As we did this we also updated a value in *overlay_network/mod.rs*. Just before closing the socket, we print the PCB with a topology number. This needs to change so we can distinguish the different topologies we tested in the logs. The user can deploy the emulator with specific configurations:

- ***/deployment/deploy -t T1*** to test the time it takes *node1* to perform path discovery. In the *ilnp_fqdn_tx()* function, we measure the time when discovery starts and when it ends. It's important here to change the number as we change the topology so we can distinguish the timings. A script in *main.rs* is used to send 50 packets to trigger 50 path discoveries.
- ***/deployment/deploy -t T2*** is used to calculate the packet overhead each time we send a single packet. A script in *main.rs* will handle this. The user simply needs to change the number of the PCB as the topology changes.
- ***/deployment/deploy -t T3*** is the same but for a 10Mbps flow.
- ***/deployment/deploy -t T4*** is to test the application throughput. Again the user only needs to change the PCB value as the topology changes.
- ***/deployment/deploy -t T5 -f hosts.txt*** this is the same test but on separate machines. It will save the logs to a separate file. Again the user needs to change the PCB value.
- ***/deployment/deploy -t T6*** is to test the latency for each packet that we send. Again *main.rs* will handle measuring the round trip time. The results are printed using a topology number. This number needs to be modified in the *main.rs* file as the topology changes. This is the case for the PCB as well.
- ***/deployment/deploy -t T7 -f hosts.txt*** is the same test as the previous one but on separate machines. It requires the same changes as the topology changes.
- ***/deployment/deploy -t A8*** is used to deploy the farming scenario. Again using a topology defined in the file. *main.rs* handles creating the fake data and sending it to the nearest sink. Again here we need to change the topology number to distinguish measurements from different topologies. We also need to define the closest sink for each node for each topology.

Please note running these scripts will append logs to the current logs used to plot the graph in the test and evaluation sections. When deploying using a topology, the user is restricted to *node* and *router* FQDNs followed by a number. This is because the deploy script uses these labels to build the network.

9.5 Software and Hardware Specifications

9.5.1 Rust

As part of the project specification, the application was implemented in Rust. Rust allows us to use low level networking functions we might expect to find in C or C++. However, unlike C, rust provides memory safety, that help us defend against attacks such as the memory buffer overflow attack. Similar to C and C++, Rust is a compiled language and we can expect efficient binaries unlike interpreters such as Python where each line runs after the next and no optimisation is done. Rust also offers libraries such as *tokio* that provide asynchronous programming and ensures thread safety at compile time. Using Rust meant we could focus on the implementation of the protocol and not too much on memory and thread management. This covers the non-functional requirement R21.

9.5.2 Operating System and Hardware

To conform with the non-functional requirement R22, the application needs to be able to run on the school lab machines. The operating system and the hardware have the following specifications:

- OS: AlmaLinux
  - Version: 9.5
  - Kernel Version: Linux 5.x
  - */proc/sys/net/core/rmem_max*: 212992
- CPU: Intel Core i5-8400 @ 2.80GHz
  - 6 cores per cpu
  - 1 thread per core
- RAM: 32 GiB
- Network Interface: Realtek RTL8111/8168/8211 (1Gbps Ethernet)

10 Performance Testing

In this section we will be testing objectives 1, 2 and 4. We will prove the protocol is able to use ILNP over IPv6 multicast and send data using the APIs. We will evaluate the performance of our protocol with 10 routers in series between 2 nodes messaging each other (as shown in Figure 33). This is the worst case scenario as there is only one path between the two nodes. If one node fails, the communication fails. It also the worst case scenario for signaling. Each router is only connected to another router. This means every router has to perform address resolution and path discovery. The third reason is this topology will allow us to understand how the application scales as we increase the number of routers.

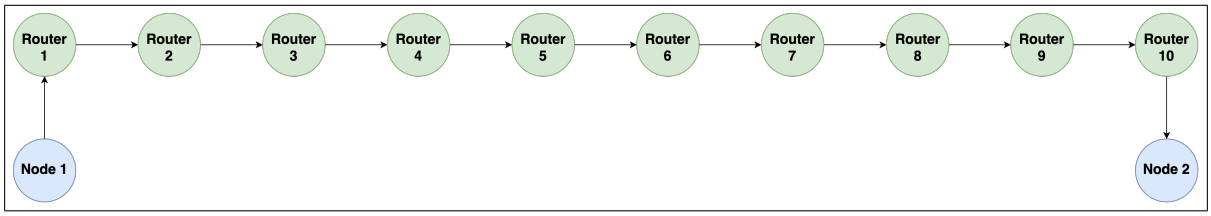


Figure 33: Network topology with 10 routers in series between 2 nodes

10.1 Path Discovery Convergence

The first performance test is measuring how long it takes for a node to discover a path after the *TTL*s expire. We will start with 1 router and finish with 10 routers in between the 2 nodes. In the *settings.toml* file, we set the address and name resolution *TTL*s to 250 seconds so discovery is done once at the begining. For path discovery, however, we set the *TTL* to 1 second so the entries would expire and a path discovery would be triggered each time we send a message. This allowed us to measure the time path discovery takes to converge. We produced 50 measurements for each topology. Figure 34 is a box chart of the elapsed time between the start and end of path discovery for *Node 1* as we increased the topology to 10 routers.

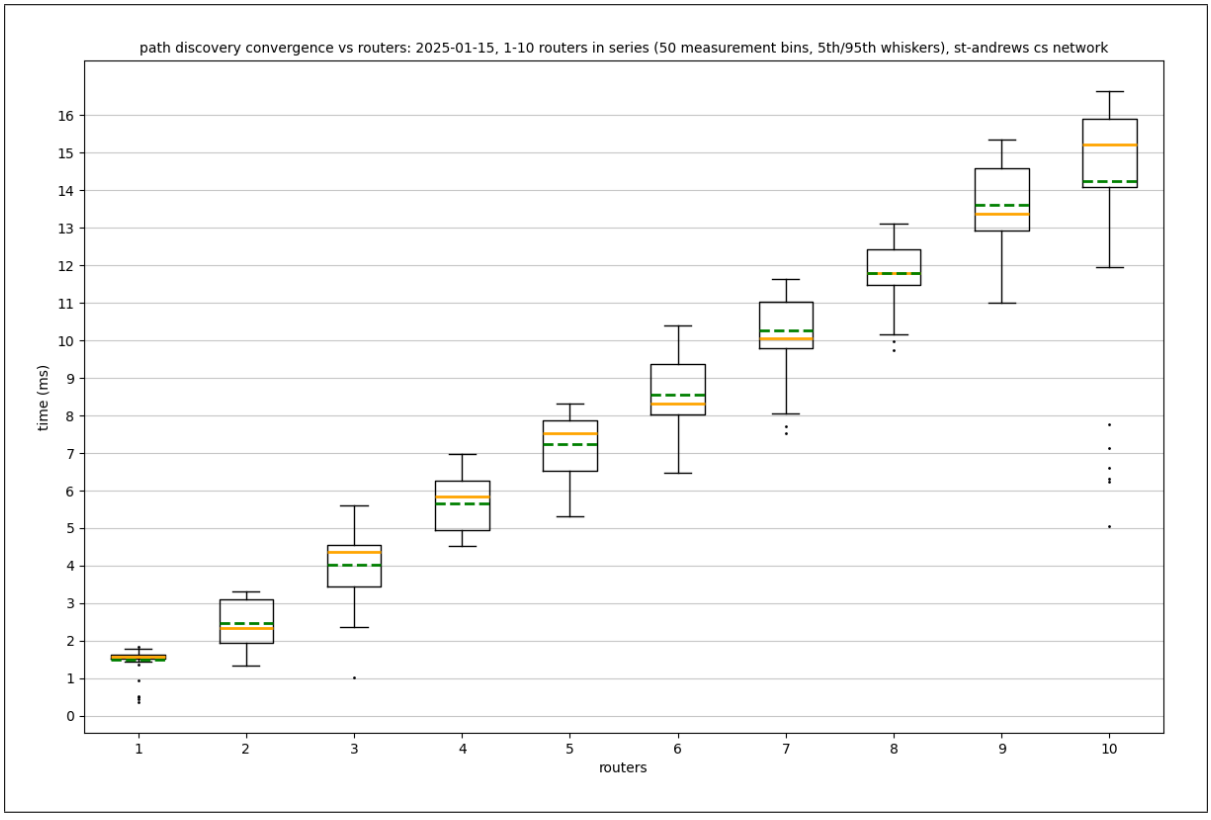


Figure 34: Box chart of path discovery convergence after *TTL*s expire as the number of routers increase

We obtained a linear result so we decided to plot a line of best fit. This is seen in Figure 35. This enabled us to discover an equation to determine the convergence time: $y = 1.55x - 0.47$. So, for example, the mean path convergence for 5 routers is given by $y = 1.55 \times 5 - 0.47 = 7.28$ milliseconds.

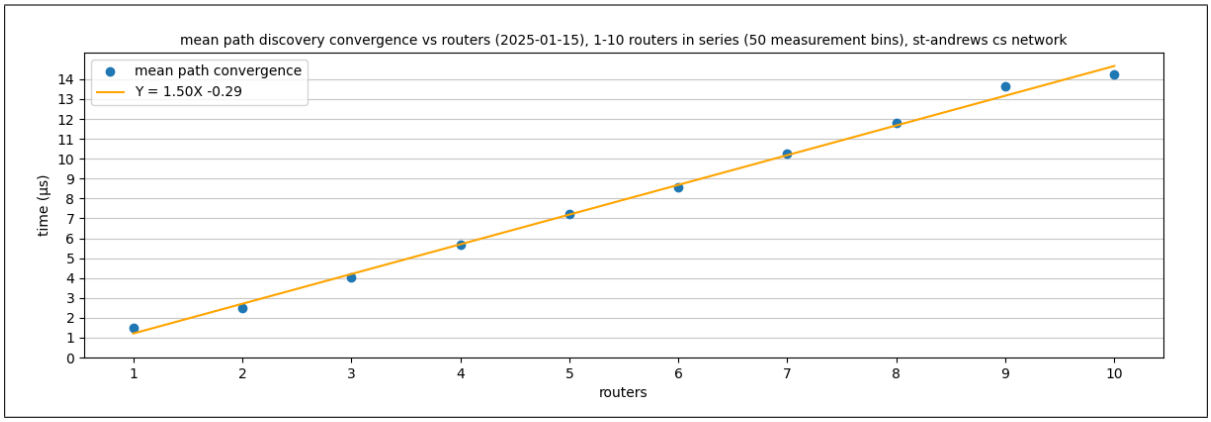


Figure 35: Line chart of the mean path convergence as the number of routers increase

10.2 Packet Overhead Single Flow

The next performance test is packet overhead. Here we will be measuring how much non-data packets are being generated to be able to send data packets. Non-data packets can be address resolution packets, name resolution packets and/or path discovery packets. Data packets are packets containing user data. A data packet traveling through routers is counted only once. For this test, we set all *TTL*s in the settings file to 1 second. This means each time we sent a packet, address resolution, name resolution and path discovery was triggered. This allowed us to calculate the packet overhead each time we send a packet with empty cache. Each time we added a router to the topology, we sent 30 data packets every 5 seconds to allow the cache to empty. After counting the number of data and overhead packets we divided the result by 30 to get an average for every topology. Figure 36 shows the average packet overhead as we increase the number of routers in the topology shown in Figure 33. The results are linear, so we can determine the packet overhead using the following equation: $y = 4x + 4$ where x is the number of routers between the nodes. For example, the packet overhead for 5 routers between 2 nodes is given by $y = 4 \times 5 + 4 = 24$ packets when caches are empty.

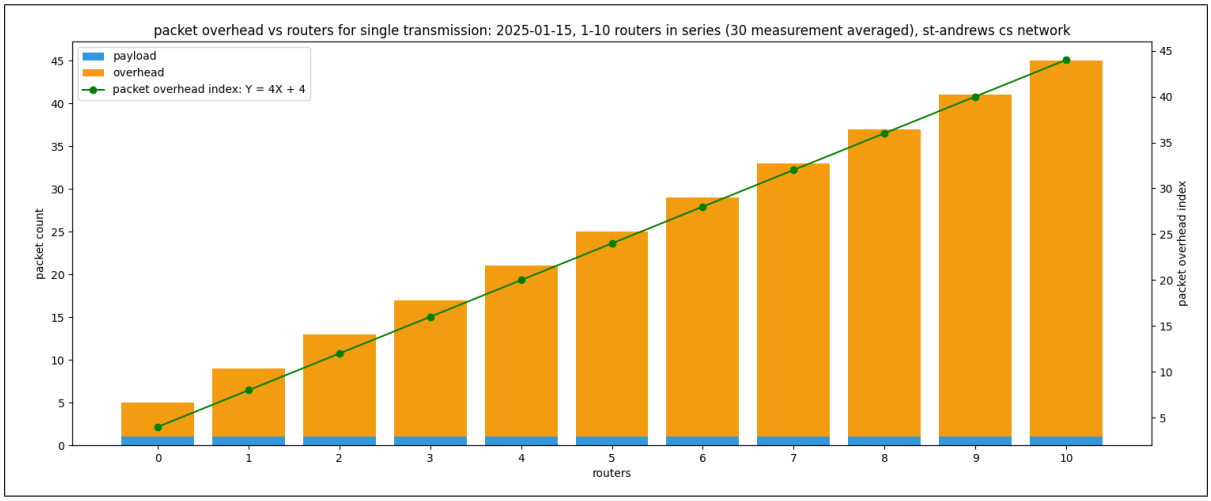


Figure 36: Bar chart packet overhead for single data packet transmissions / Line chart mean packet overhead for a single tranmission as the number of routers increase

10.3 Packet Overhead Continuous Flow

Packet overhead is required to discover addresses, names and paths. Once this information is known it can be retained in cache using *TTL*s. The larger the *TTL* the less likely something needs to be discovered again and thus reducing packet overhead. However, the higher the *TTL*, the less likely we will notice a node failure. For this test we set all the *TTL*s to 1 second again. We sent 10Mb worth of user data in one second, waited for the TTL to expire and repeated the process 30 times and averaged the results. This was done for each router that we added to the topology shown in Figure 33. Figure 37 demonstrates the packet overhead of a 10Mbps flow. We can see that we can afford to send large amounts of data without too much packet overhead. However, the packet overhead does still increase as the number of router increases.

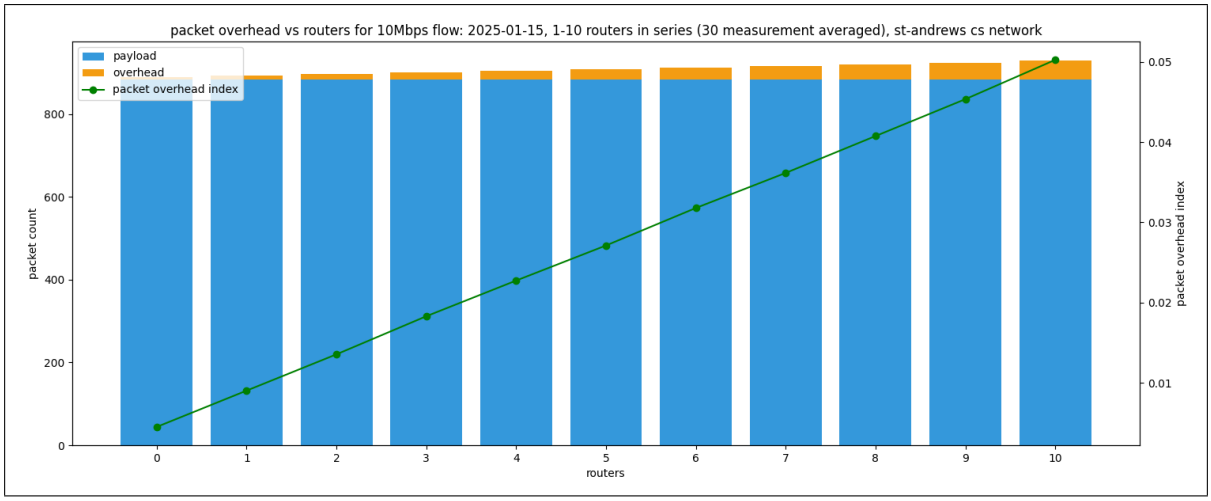


Figure 37: Bar chart packet overhead for data packet flow / Line chart mean packet overhead for data packet flow as the number of routers increase

10.4 Packet Throughput

The next performance test is pushing the application to the limit and seeing the maximum data rate we can achieve. For this, we created a 30 second loop that sent as many packets as the rust socket would allow us. For this test, we set high *TTL*s so discovery would only be done once. This is so we could purely test the throughput of the application. Figure 38 plots the throughput of the sender. We then attempted to receive as many packets as possible on the receiving node. This is shown in Figure 39. We repeated the experiment while increasing the number of routers to give us the topology in Figure 33. In these figures, we ran each node on a separate linux machine in the same LAN network. From Figure 38 and 39 we can see the application is able to send and receive at the same rate giving a minimum of 900Mbps throughput. This is constant as we increase the number of routers between the nodes.

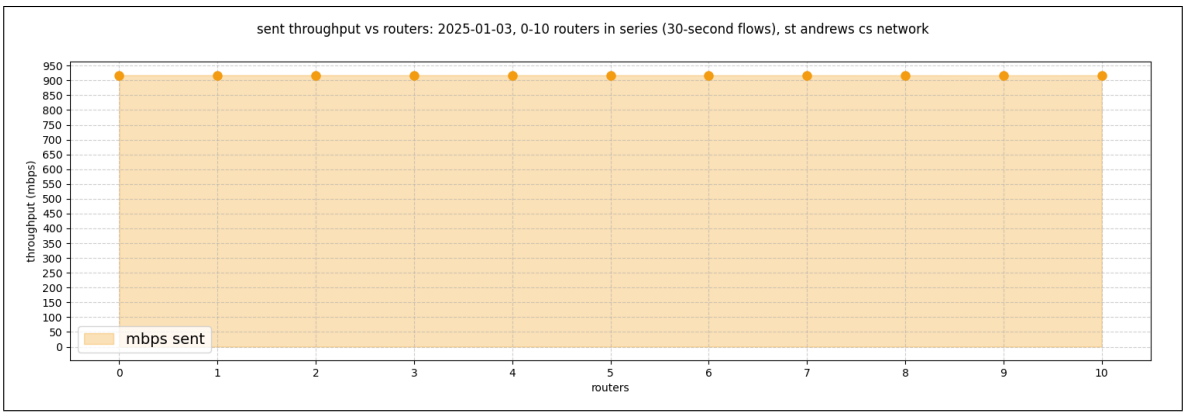


Figure 38: Line chart of the outgoing throughput when running nodes on separate machines

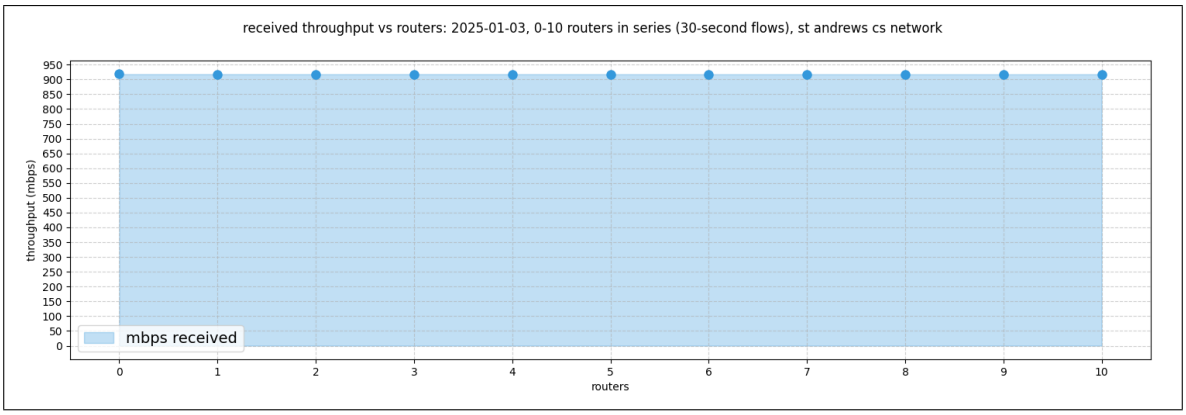


Figure 39: Line chart of the incoming throughput when running nodes on separate machines

We then ran multiple instances of the application on the same instance to test the throughput. This is seen in Figure 40 and 41. The application uses *IPV6_MULTICAST_LOOP* set to *true* to receive packets from other instances [32]. As we can see from Figure 41 the receiving feature of the application degrades as we increase the number of instances running on the same machine. When we get to five routers, we reach a throughput of around 1Gbps. At 10 routers, it goes as low as 200Mbps. This is a performance issue. After some investigation, we noticed the UDP buffer queues are overflowing and dropping packets. This can be seen by running "*watch -n 0.1 cat /proc/net/udp6*". We concluded this to mean that our receiver in the application is not consuming packets fast enough to empty the UDP queues at the kernel level.

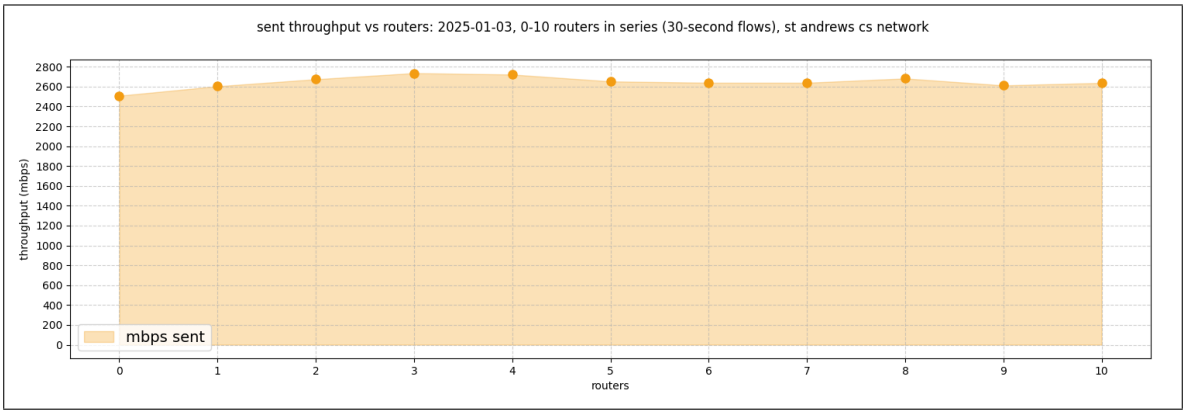


Figure 40: Line chart of the outgoing throughput when running nodes on the same machine

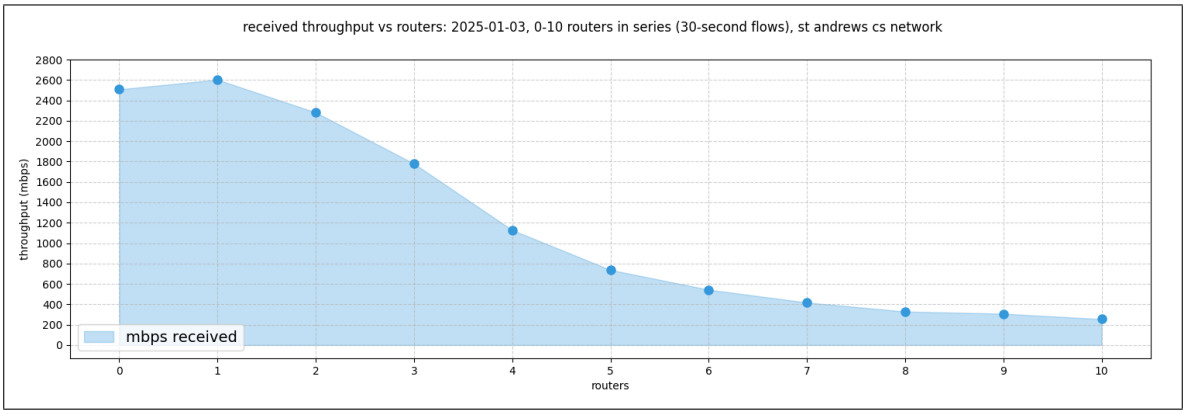


Figure 41: Line chart of the incoming throughput when running nodes on the same machine

In Figure 42, we were able to prove that the instances are competing for a shared resource due to the rise in *EAGAIN futex* errors. Examples of such errors are available in Figure 43. With more time, we would have liked to investigate the cause of this.

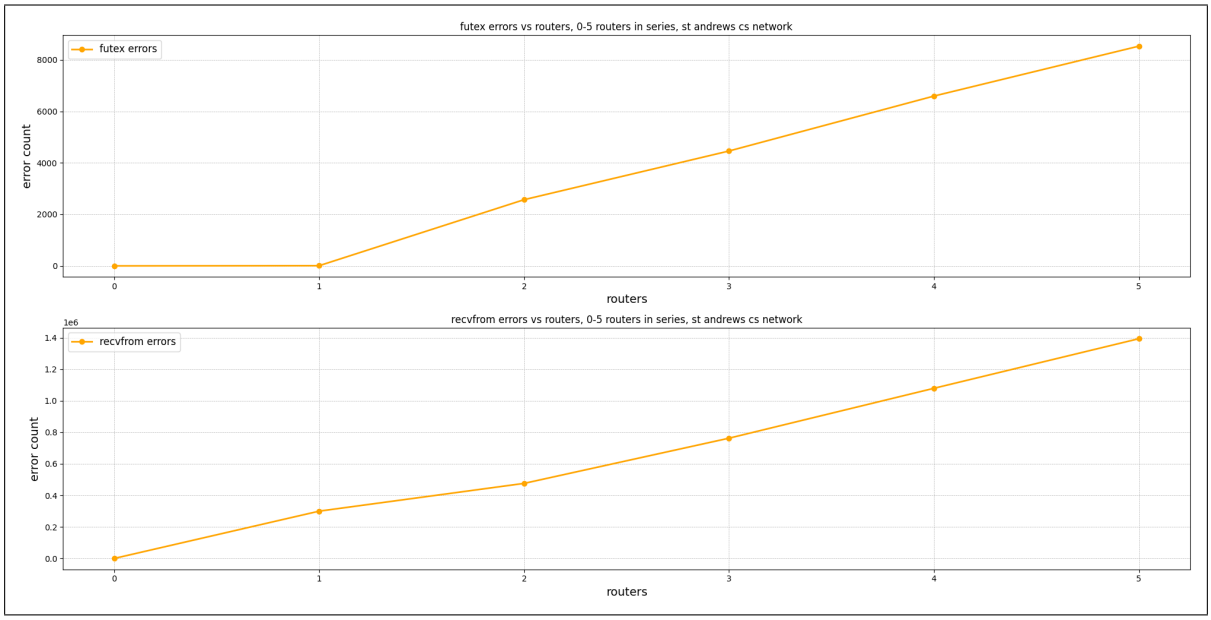


Figure 42: System call errors as we increase the number of routers

```
jlbl@pc7-013-l:~$ strace -f -s 256 -p 1 2>&1 | awk '/futex/ && /-1/ { print }'
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     6] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     5] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     5] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     5] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     5] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     5] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     5] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     5] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
[pid     4] <... futex resumed>)        = -1 EAGAIN (Resource temporarily unavailable)
```

Figure 43: Logs recovered from running strace in the podman containers

10.5 Packet Latency

The last performance test in this section is a latency test. We send a payload containing a sequence number and starting time in microseconds. Upon receiving the payload we check the sequence number is correct and calculate the elapsed time giving us the round trip time of the packet. We were getting varied results, so we decided to repeat this 2000 times and draw a box chart. This was repeated as we increased the number of routers to get the topology in Figure 33. The *TTL*s in the *settings.toml* were set to 60 seconds and we sent a message every 5 seconds 2000 times. We choose 5 seconds to give a chance for the queues to empty and get an accurate round trip time for a single packet. We expect to see delays due to address resolution, name resolution and path discovery. Figure 44 is box chart of 2000 round trip time (RTT) measurements. The round trip time starts just above 500 microseconds and ends at around 5 milliseconds at 10 routers. It's important to note that for each topology, the first round trip time was significantly higher. This is because initially all the caches are empty so discovery needs to be done for every node. We notice this also at the IPv6 layer when the first latency reading from *ping* is significantly higher when the destination machine is not in the source machine's ARP table. Because of this, we excluded the first reading of the 2000 measurements for each topology. By including it, the boxes were squashed and we could not analyse the trend in the latency.

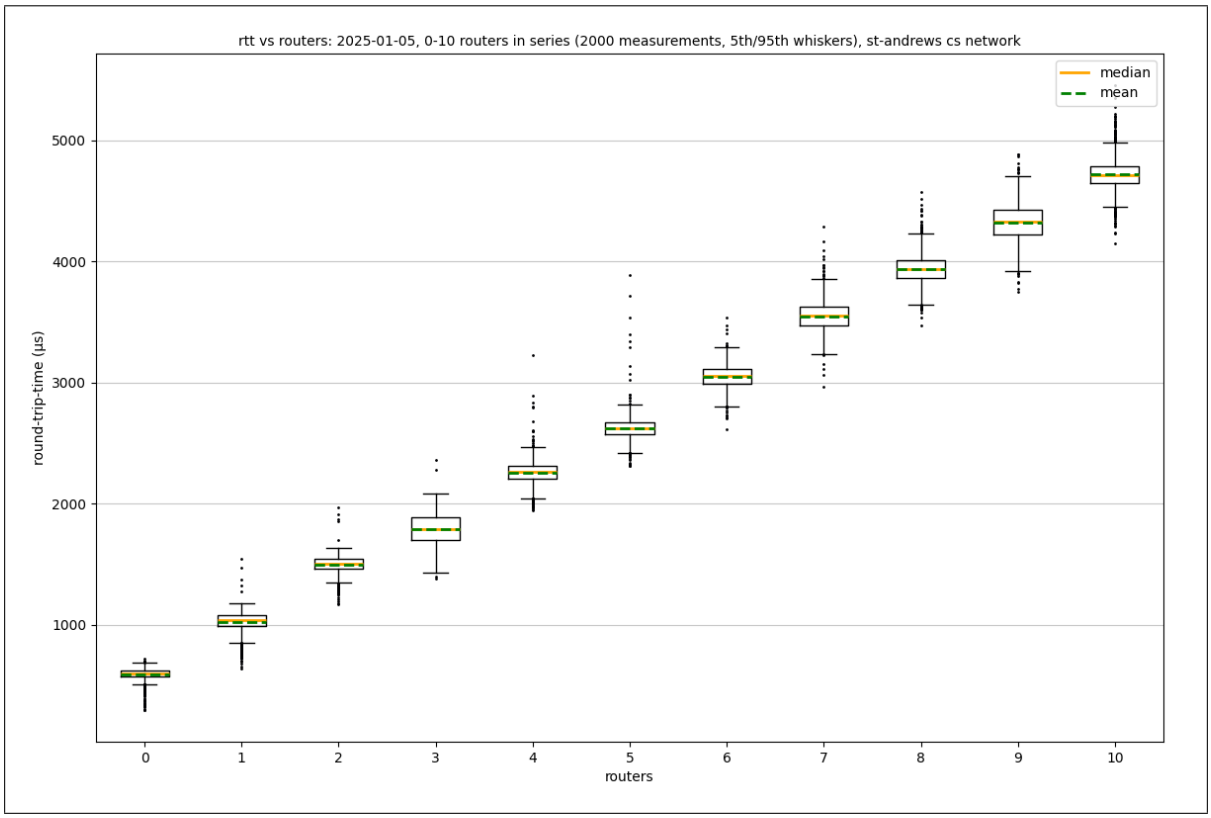


Figure 44: Box chart of the packet round trip times when running nodes on separate machines

Figure 45 is the same box chart but when running the different nodes and routers on the same machine. This chart shows lower round trip times. It is just below 500 microseconds with no routers and ends just before 3.5 milliseconds at 10 routers. From these two box charts we can see a linear increase as we increase the number of routers between the two nodes. Because of this, we decided to plot the line of best fit for the mean latency. This is seen in Figure 46 for separate machines and 47 for the same machine. The figures allowed us to find an equation that determines the latency based on the number of routers. When running the nodes on separate machines, the latency can be given using: $y = 207x + 95$ where x is the number of routers in the shortest path between the two nodes. Similarly, when running nodes on the same machine, the latency can be given using: $y = 124x + 51$ where x is again the number of routers in the shortest path. Both equations give the latency in microseconds.

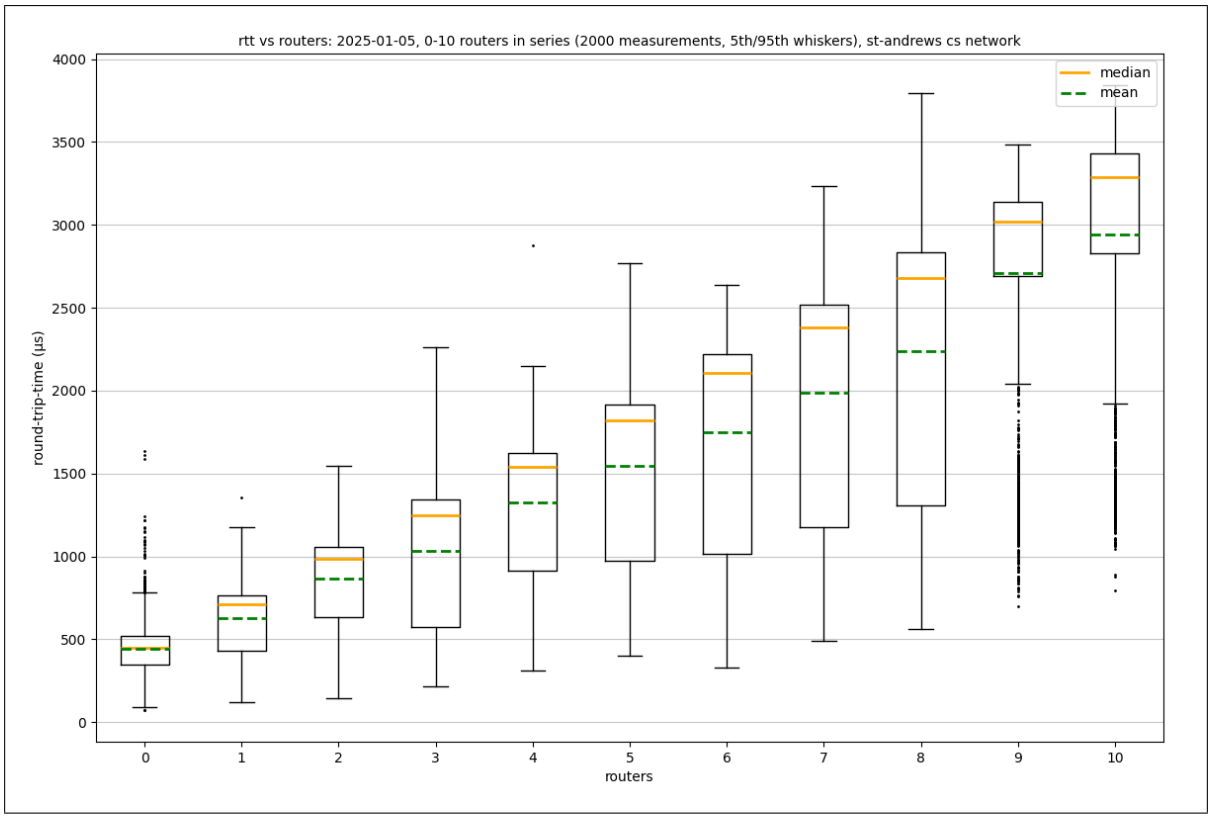


Figure 45: Box chart of the packet round trip times when running nodes on the same machine

Figure 46: Line chart of mean the packet latency when running nodes on separate machines

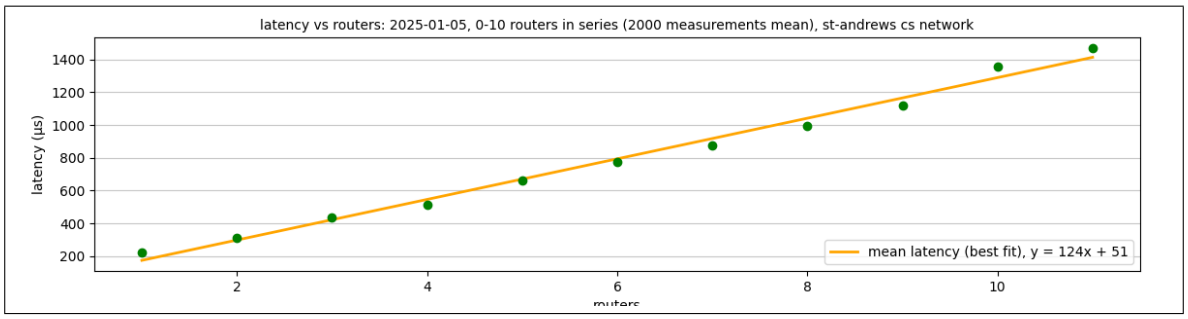


Figure 47: Line chart of mean the packet latency when running nodes on the same machines

10.6 Performance Testing Conclusion

To conclude the performance testing section, we determined that the maximum path convergence time in a 10 router topology is 15 milliseconds. The maximum packet overhead, each time the TTL expire, is 44 packets for a single data packet but also 44 for a continuous data packet flow. On separate machines the maximum throughput is 900Mbps. It is 200Mbps on a single machine. In a ten 10 router topology, the maximum latency we can expect is 5 milliseconds on separate machines and 3.5 microseconds on the same machine. We were also able to establish equations to predict network properties based on the number of router, where x is the number of routers:

- Convergence Time: $y = 1.55x - 0.47$ milliseconds.
- Packet Overhead: $y = 4x + 4$ packet overhead.
- Separate Machine Packet Latency: $y = 207x + 95$ microseconds.
- Same Machine Packet Latency: $y = 124x + 51$ microseconds.

In the design and implementation section we were able to prove that the protocol uses ILNP infrastructure and is built on top of IPv6 multicast. This covers objective 1 and 2. In this section, we have been able to show these objectives in action. We were also able to prove that the user is able to use APIs to send traffic though the emulated network with reasonable performance. This covers objective 4. We were able to prove data can be sent using *NID* and *FQDN* through a customised topology using routers by calling the application's APIs. This proves the user requirements R1, R2, R3, R4, R5 and R6. It also proves the functional requirements R8, R9, R10, R11, R12, R13, R14, R16, R17, R18, R19 and R20.

To be able to simulate ILNP without making changes at the kernel level, we needed to encapsulate packets using an extra ILNPv6 (which is the same size as an IPv6 header). Because of this, additional overhead is expected. On the school lab machines, using *ping* we found the average latency to be around 200 microseconds between each machines. Using *iperf3*, we found the throughput to be just below 950Mbps. We see the same latency in our results between 2 nodes (no routers). We also see similar throughput results when running the application on separate machines. Using our equation, the path convergence for a 100 router topology would be $1.55 \times 100 - 0.47 = 154$ milliseconds. This is still under a second and so is acceptable for this application.

11 Evaluation and Critical Appraisal

11.1 Topology Analysis

In objective 3, we aimed to allow users to setup custom networks with custom traffic flows. We also said users should be able to send traffic through these emulated network using the ILNP API. In this section we prove that we can create custom network topologies and setup custom data flows using the JTP API built on top of our ILNP API. Additionally, we will test path stretch, which is the index that measures how well a node sends a packet using the shortest path possible. For this, we concocted a scenario around IoT devices for agriculture. After a hike in my neighbourhood, we have designed farm fields inspired by the fields seen on the walk. These designs can be seen in Figure 48.

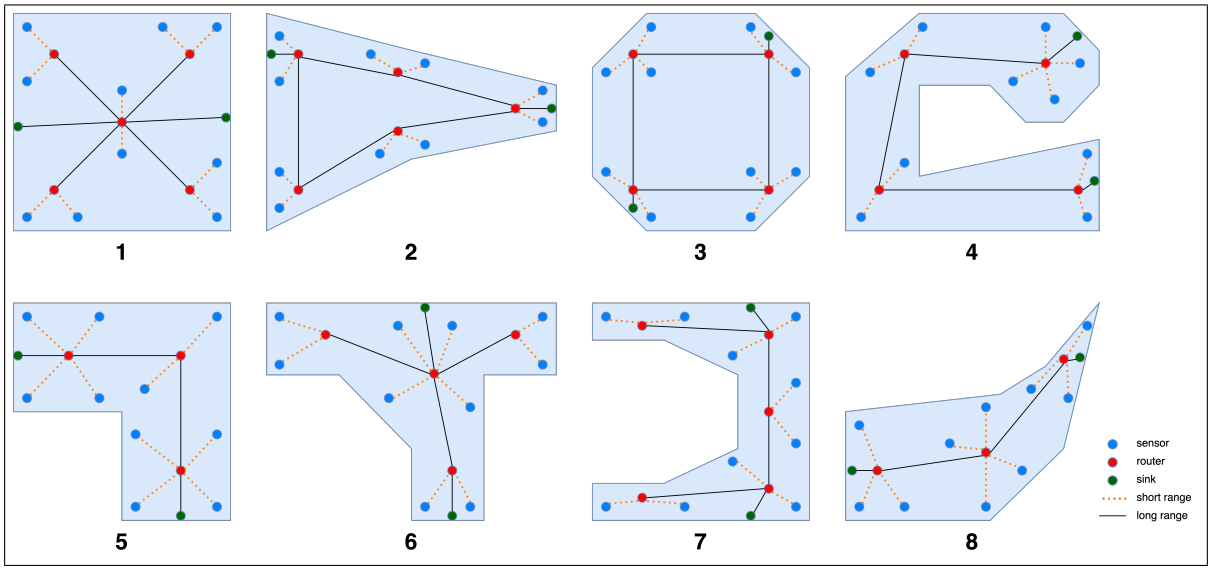


Figure 48: Farm fields and IoT network topologies

On top of these fields, we have designed an IoT layout for measuring temperature, humidity and soil moistures in different parts of the fields. Sensors (blue dots) use low power short range wireless networks to connect to routers (red dots). Routers and Sinks are connected using high powered long range networks. Sinks are nodes on the edge of the network that is connected to the rest of the internet. In a real life scenario, this node would be responsible for sending the measurements to the data centers. Note we do not claim this setup to be the most optimised for collecting farming data, this is simply to test our objectives and the performance of our emulated network using different topologies. For this experiment, we decided to send fake temperature, humidity and soil moisture readings every five seconds. In a real life scerario, every 5 minutes would be more appropriate, however this would have taken too long to run. We decided to use 10 nodes and 2 sinks for each topology to simplify the monitoring. Figure 49 is the setting file that we used. Notice all the *TTL*s are set to 1 second. This means discovery will need to be done for each measurement we send. We decided to this for two reasons, the first is this will prove that the nodes can be mobile as discovery is done each time we send a measurement. Secondly, this will allow us to measure path discovery and calculate the path stretch. Each sensor node begins with a random temperature, humidity and soil mosture within a given range. They then slowly increase these values at different steps until the maximum is reached. The values then decrease until the minimum is reached. This is repeated until each device has sent 20 measurements. This was necessary to generate the fake data. In Figure 50 we plotted the measurements logged by the sink nodes which are measurements sent by sensor nodes through the first topology in Figure 48. This proves we were able to setup a custom topology and send data through it using our emulated ILNP network. We only show the fake data charts of one topology as the charts look similar for all the other topologies. As we can see, the sinks have received farming data from all the devices.

```
[network]
MTU = 1412
ND_RTO_MS = 1
ND_RETRANSMIT_LIMIT = 3
ND_TTL_S = 1
ND_CACHE_SIZE = 100
DNS_TTL_S = 1
AD_HOC_TIMEOUT_MS = 100
AD_HOC_RTO_NS = 100
AD_HOC_TTL_S = 1
AD_MAX_HOPS = 15
```

Figure 49: Settings file for farming scenario

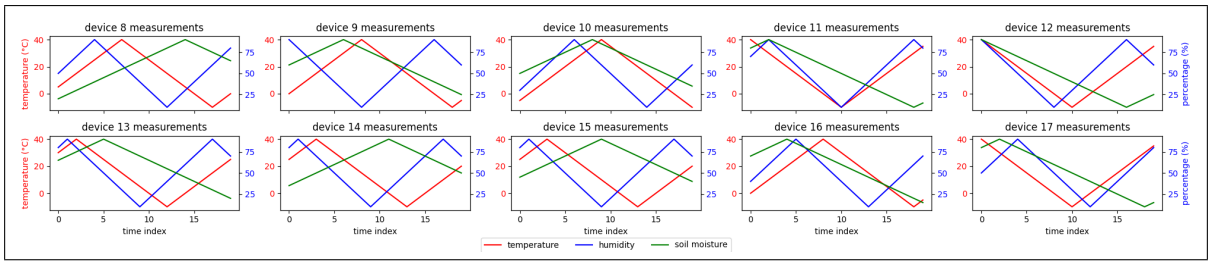


Figure 50: Received measurements from the both sinks in first topology

In each topologies, *node1* and *node2* FQDNs are reserved for the sinks. So each topology will have two sinks. For each topology, the closest sink to each sensor node is defined by the user at the application level. This is simply because sinks are not part of the ILNP protocol, this is an application level design. If a device fails to discover its default sink, it is allowed to send the packet to the second sink through the emulated network. We will be measuring two things here:

- $sink\_stretch = \frac{nb\_total\_default\_sink}{nb\_total\_measurement\_sent}$
- $path\_stretch = \frac{nb\_total\_packet\_forward}{nb\_total\_packet\_forward\_shortest\_path}$

Figure 51 is the topology configuration for the first topology in Figure 48. *node1* and *node2* are our sinks. *router3* is the center router. All routers have their own connection to the center router. Sensor nodes are grouped into short range networks with a router to route their packets through the long range connections. At the application layer, we told *node 1-7* to send to *node1* (the first sink) and *node 8-12* to send to *node2* (the second sink). This emulates the first network topology. Using this example, we know each device needs to send 20 measurements. With 10 devices (12 nodes minus the sinks) we should get $10 \times 20 = 200$ measurements. Both sinks should receive 100 measurements each. If we receive 100 at the first sink then the sink stretch will be $sink_1\_stretch = \frac{100}{100} = 1$. If we receive 75 and the other sink receive 125, then 25 measurements would have failed to send to the first sink and was sent to the second sink. This would give $sink_1 = \frac{75}{100} = 0.75$ and $sink_2 = \frac{125}{100} = 1.25$. We can plot this as we go through the topologies. Secondly the sensors on the edge of the field have to travel through 2 routers to get to the sink. With 20 measurements, 8 devices, 2 routers, we should measure $20 \times 2 \times 8 = 320$ forwarded packets. The other 2 sensors in the middle only need to travel through 1 router giving $20 \times 1 \times 2 = 40$ forwarded packets. In total, $320 + 40 = 360$ packets should have been forwarded if the shortest path was used. Using this logic, we were able to make the top half of the table in Figure 52 which are the expected values in each topology. Notice in the 6th topology, 8 sensors are closest to the first sink. This is why the first sink is expected to receive $20 \times 8 = 160$ measurements. And as a result we expect the second sink to receive 40 measurements. We ran each topology while logging the PCB and the Sinks. After processing these logs we got the results in the bottom half of the table in Figure 52. We decided to plot the results using $stretch = \frac{actual}{expected}$. The first and second sink stretch as well as the path stretch are plotted in Figure 53, Figure 54 and 55 respectively. From these charts we can see the $sink_1$, $sink_2$ and path stretch is always 1. This means, using 20 measurements for each of the 8 toplogies, the sensor's default sink is always discovered and the shortest paths to those sinks are always used.

```yaml
routers: 5
nodes: 12

links:
  - network: [node1, router3]
  - network: [node2, router3]

  - network: [router1, router3]
  - network: [router2, router3]
  - network: [router4, router3]
  - network: [router5, router3]

  - network: [router1, node3, node4]
  - network: [router2, node5, node6]
  - network: [router3, node7, node8]
  - network: [router4, node9, node10]
  - network: [router5, node11, node12]
```

Figure 51: The topology yaml file for the first topology

		1	2	3	4	5	6	7	8
Expected	Sink1	100	100	100	120	100	160	100	100
	Sink2	100	100	100	80	100	40	100	100
	Path	360	320	320	280	240	280	320	280
Actual	Sink1	100	100	100	120	100	160	100	100
	Sink2	100	100	100	80	100	40	100	100
	Path	360	320	320	280	240	280	320	280

Figure 52: Expected and Actual values for number of received measurement at each sink and the number of packets forwarded in each topology

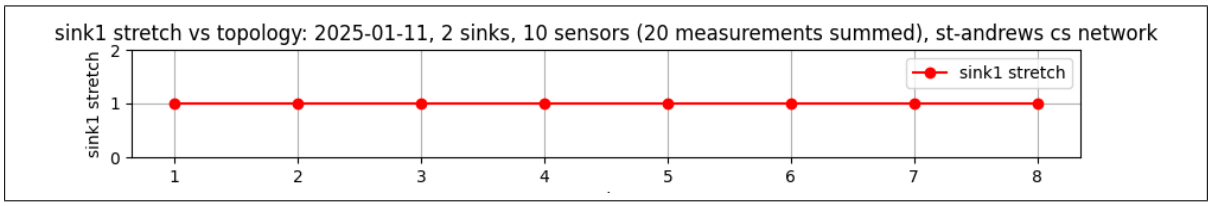


Figure 53: Line chart of the Sink1 stretch for each topology

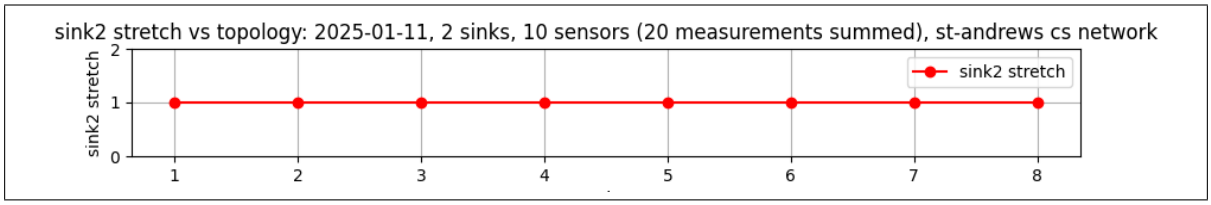


Figure 54: Line chart of the Sink2 stretch for each topology

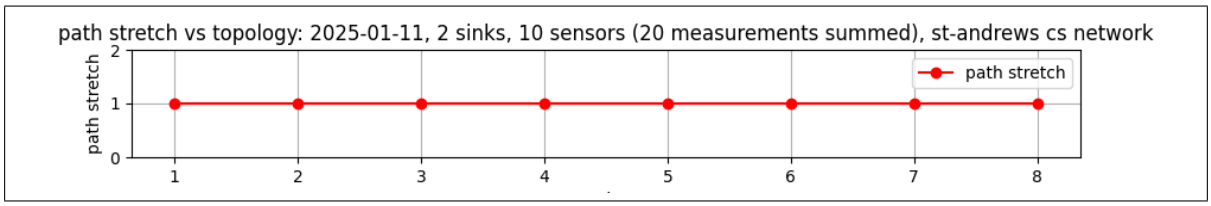


Figure 55: Line chart of the path stretch for each topology

Here we have demonstrated that the network emulator is capable of having custom network topologies and is able to send information through the topologies. This covers R1, R2, R8, R17 and R20. We have also demonstrated that the information travels the most efficiently through the topologies increasing the performance of our solution.

11.2 Multi-Homed Topology Analysis

In attempt to complete objective 5 and requirements R7 and R15, we allowed the nodes to be multi-homed. This would mean they would eventually be able to perform a "make before break" and seamlessly switch between networks. Unfortunately, we didn't have the time to allow the nodes to switch between networks. Figure 56 is an example topology. Notice we have 2 nodes and a sink that are multi-homed. We tested this topology with the same logic as above. All measurements were sent to their closest sinks. However the shortest paths to the sinks weren't always taken. For example, we noticed the bottom left sink would send the measurements through the 3 bottom routers before arriving to the bottom sink. The path stretch therefore went above 1. Because of time constraints, we could not investigate the cause of this. Despite this, all measurements were received form all devices to the correct sinks.

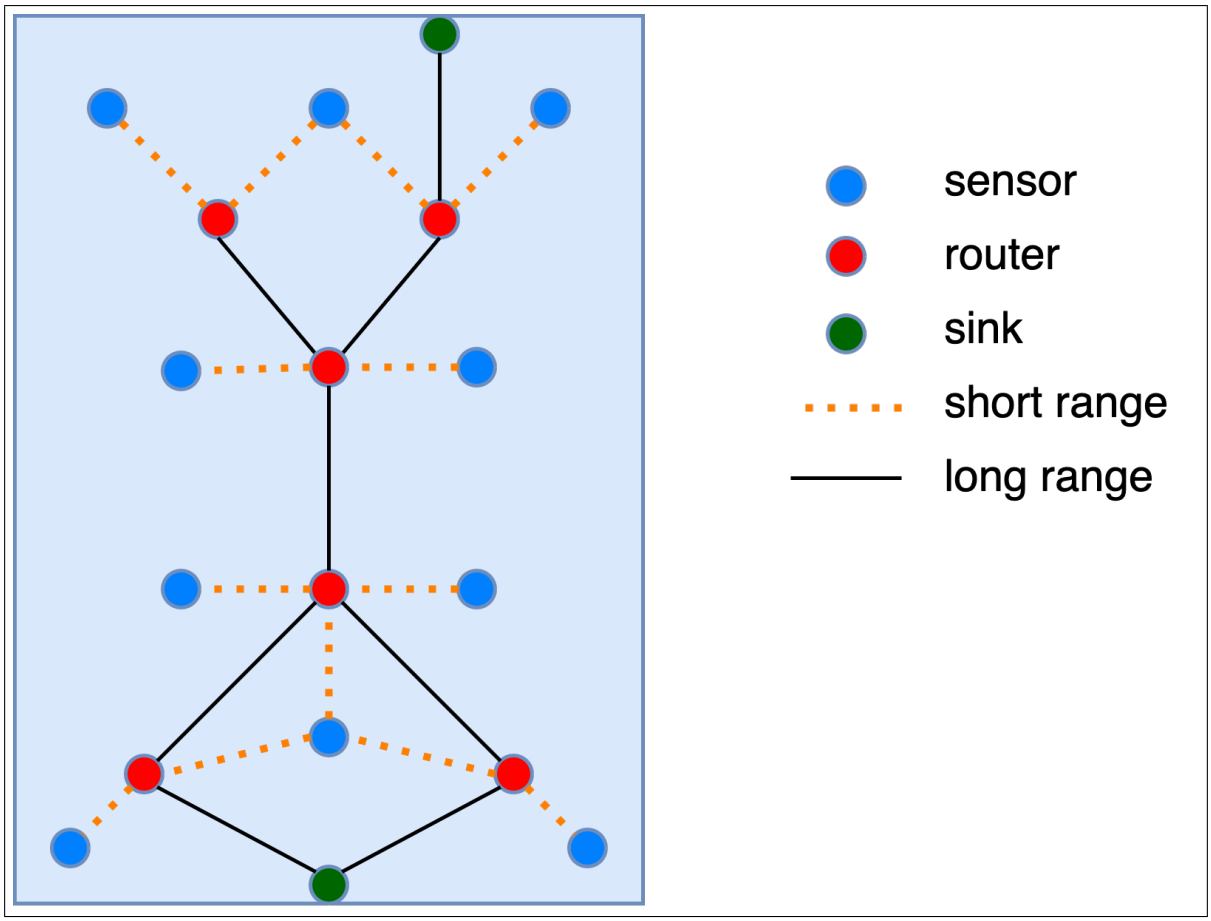


Figure 56: Farm field with a multi-homed IoT topology

11.3 Room for Improvement

To improve this project, the domain names could use the *.ilnp* extension. So that if this project is expanded and used in the wide area network, the domain names would not clash with real life domain names. A second improvement could be to build a graphical interface to build different network topologies using drag and drop. Furthermore, the project can be extended to allow nodes to travel through different network and still use the shortest path when multi-homed. The JTP transport protocol can also be expanded to include more headers and provide a more reliable transmission. For instance, it can use sequence numbers and trigger a retransmission when a UDP packet is lost. To strengthen the claims in this report we can test a node against a netcat script to test individual features. Furthermore, an attacker can create a fake router and respond to router request with a router response that has a hop count of 0. This will tell the node that the fake router is connected to the network the node is trying to send data to. This would fool the node in forwarding all packets through the fake router and allow the attacker to snoop. The project can be improved by adding a security layer. Finally, to reduce the latency further, the application could increase the *TTL*s in the tables each time it receives a packet. This would reduce the need to perform discovery. With more time we would have liked to implement this change.

12 Conclusions

To conclude, users are able to setup custom networks using nodes and routers. They are able to define the *NID* and *FQDN* for each node. Users can also use APIs to send and recieve packets using either the destination *NID* or *FQDN*. This means we completed all the user requirements except for R7 which allows the nodes to travel across the networks. As long as all networks are connected by routers, a packet can reach its destination. Nodes are able to discover each other's identify and location on the emulated network. This means we have completed all the functional requirements except for R15 which states nodes should be able to move across networks without breaking the connection. We have completed all the non-functional requirements as the application runs using rust, runs on the school lab machines and uses ILNP emulated over multicast IPv6. We have therefore completed our primary objectives (1, 2, 3, 4) and made a start at secondary objectives (5).

The solution showed reasonable path discovery convergence times. It also showed reasonable packet overhead which doesn't storm the emulated network. It showed we were able to achieve similar throughput to *iperf3* except when running the nodes on the same machine. Future work could explore the cause of this and provide a fix. The application also displayed similar latencies to *ping* when the comparison was appropriate (2 nodes). We were also able to establish methods to anticipate the network's delay and overhead before deployment.

The results conclude that, despite not having mobile nodes, the application can be used to experiment with the ILNP addressing architecture without the overhead of setting up OS kernel installations. This was the aim of the project.

References

[1] M. Komu, M. Sethi, and N. Beijar, "A survey of identifier–locator split addressing architectures," *Computer Science Review*, vol. 17, pp. 25–42, 2015.

[2] R. Atkinson, S. Bhatti, and S. Hailes, "A proposal for unifying mobility with multi-homing, NAT, & security," in *Proceedings of the 5th ACM international workshop on Mobility management and wireless access*, pp. 74–83, 2007.

[3] R. Yanagida and S. N. Bhatti, "Seamless Internet connectivity for ubiquitous communication," in *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers*, pp. 1022–1033, 2019.

[4] R. Atkinson and S. Bhatti, "Identifier-locator network protocol (ILNP) architectural description," RFC 6740, November 2012.

[5] M. Weiser, "Some computer science issues in ubiquitous computing," *Commun. ACM*, vol. 36, p. 75–84, July 1993.

[6] P. Manzoni, D. Ghosal, and G. Serazzi, "Impact of mobility on TCP/IP: an integrated performance study," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 5, pp. 858–867, 1995.

[7] J. Day and H. Zimmermann, "The OSI reference model," *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334–1340, 1983.

[8] H. Schulzrinne, E. Schooler, J. Rosenberg, and M. J. Handley, "SIP: Session Initiation Protocol." RFC 2543, Mar. 1999.

[9] T. Kwon, M. Gerla, and S. Das, "Mobility management for VoIP service: Mobile IP vs. SIP," *IEEE Wireless Communications*, vol. 9, no. 5, pp. 66–75, 2002.

[10] M. Kulin, T. Kazaz, and S. Mrdovic, "SIP server security with TLS: Relative performance evaluation," in *2012 IX International Symposium on Telecommunications (BIHTEL)*, pp. 1–6, IEEE, 2012.

[11] C. Perkins, "IP Mobility Support for IPv4, Revised," RFC 5944, November 2010.

[12] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[13] D. A. Joseph, "Mobility Support in IPv6," *Association for Computing Machinery*, November 1996.

[14] D. B. Johnson, J. Arkko, and C. E. Perkins, "Mobility Support in IPv6." RFC 6275, July 2011.

[15] K. Seo and S. Kent, "Security Architecture for the Internet Protocol." RFC 4301, Dec. 2005.

[16] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, "The Locator/ID Separation Protocol (LISP)." RFC 6830, Jan. 2013.

[17] R. Atkinson and S. Bhatti, "ICMP locator update message for the identifier-locator network protocol for IPv6 (ILNPv6)," RFC 6743, November 2012.

[18] R. Atkinson and S. Bhatti, "IPv6 Nonce Destination Option for the Identifier-Locator Network Protocol for IPv6 (ILNPv6)," RFC 6744, November 2012.

[19] D. Phoomikiattisak and S. N. Bhatti, "IP-layer soft handoff implementation in ILNP," in *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture*, pp. 1–6, September 2014.

[20] R. Atkinson, S. Bhatti, and S. Rose, "DNS Resource Records for the Identifier-Locator Network Protocol (ILNP)." RFC 6742, Nov. 2012.

[21] P. Mockapetris, "Domain names - implementation and specification." RFC 1035, Nov. 1987.

[22] S. R. Das, C. E. Perkins, and E. M. Belding-Royer, "Ad hoc On-Demand Distance Vector (AODV) Routing." RFC 3561, July 2003.

[23] "User Datagram Protocol." RFC 768, Aug. 1980.

[24] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460, Dec. 1998.

[25] Internet Assigned Numbers Authority, "Protocol Numbers," 2024. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[26] J. Postel, "Internet Control Message Protocol." RFC 792, Sept. 1981.

[27] R. Atkinson and S. Bhatti, "Address Resolution Protocol (ARP) for the Identifier-Locator Network Protocol for IPv4 (ILNPv4)." RFC 6747, Nov. 2012.

[28] W. A. Simpson, D. T. Narten, E. Nordmark, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)." RFC 4861, Sept. 2007.

[29] B. Yousefi and F. Ghassemi, "An Efficient Loop-free Version of ADOVv2," *CoRR*, vol. abs/1709.01786, 2017.

[30] J. Goulding, "How to set the socket option SO_REUSEPORT in Rust," 2016. https://stackoverflow.com/questions/40468685/how-to-set-the-socket-option-so-reuseport-in-rust.

[31] R. Documentation, "Rust: modular_bitfield_msb," 2025. https://docs.rs/modular-bitfield-msb/latest/modular_bitfield_msb/.

[32] Rust Programming Language, "Rust libc ipv6_multicast_loop." https://docs.rs/libc/latest/libc/constant.IPV6_MULTICAST_LOOP.html, 2025.