Ubiquitous Communication for the Internet of Things

A Software Emulation of ILNP-Enabled IoT devices, with an Experimental Evaluation

Author: Daniel Vallance
Supervisor: Saleem Bhatti

University of St Andrews

Senior Honours Project submitted to the
Department of Computer Science of the University of St Andrews

27th March 2023

1 Abstract

In 1993, Weiser proposed a vision of the future of computing called "ubiquitous computing"[40], in which users have much wider access to self-configuring computing power. Current trends in IoT technology do widen the user's access to computing power so they do partially qualify as ubiquitous computing, however their usage of IP prevents them from implementing ubiquitous communication which is a prerequisite for ubiquitous computing. ILNP is an alternative network layer protocol that uses separate Identifier and Locator values to facilitate ubiquitous communication in a way IP cannot. This project includes a software emulation of ILNP-enabled IoT devices which runs as an overlay network, an API for the emulation and a set of applications that model scenarios which require ubiquitous communication in the form of mobility and multipath communication. Finally, the ILNP emulation's performance in these scenarios is compared to that of a similar IPv6 emulation in order to determine the relative suitability of ILNP for ubiquitous computing.

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 20,259 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.

This report © 2023 by Daniel Vallance is licensed under CC BY 4.0
(https://creativecommons.org/licenses/by/4.0)

Contents

1 Abstract **1**

2 Declaration **2**

3 Introduction **6**
3.1 Objectives . . . 7
3.1.1 Primary Objectives . . . 7
3.1.2 Secondary Objectives . . . 7
3.2 Project Overview . . . 8

4 Context Review **9**
4.1 Ubiquitous Systems . . . 9
4.2 ILNP . . . 10
4.3 IoT . . . 11

5 Requirements Specification **13**
5.1 Functional Requirements . . . 13
5.1.1 Functional Requirements of Objective P1 . . . 13
5.1.2 Functional Requirements of Objective P2 . . . 14
5.1.3 Functional Requirements of Objective P3 . . . 14
5.1.4 Functional Requirements of Objective S1 . . . 14
5.1.5 Functional Requirements of Objective S2 . . . 15
5.2 Non-Functional Requirements . . . 15
5.2.1 Non-Functional Requirements of Objective P1 . . . 15
5.2.2 Non-Functional Requirements of Objective P2 . . . 15
5.2.3 Non-Functional Requirements of Objective P3 . . . 16
5.2.4 Non-Functional Requirements of Objective S1 . . . 16
5.2.5 Non-Functional Requirements of Objective S2 . . . 16

6 Ethics **17**

7 Emulation Design and Implementation **18**
7.1 Overlay network . . . 18
7.1.1 Network Layer . . . 18
7.1.2 Transport Layer . . . 19
7.1.3 Link Layer . . . 20
7.1.4 Overall Overlay Network Stack . . . 21
7.2 Router Advertisements . . . 21
7.3 Soft Vertical Handoff . . . 23
7.4 Handling of Own Locators . . . 25
7.5 Neighbour Advertisements . . . 28
7.6 Sending . . . 31
7.7 Generation of Node Identifiers and Locator Values . . . 33
7.8 Concurrency . . . 33
7.8.1 Choice of Concurrency Module . . . 33

7.8.2 Asyncio Constructs Used . . . 34
7.9 Control Plane and API . . . 35
7.10 Management . . . 37
7.11 Configuration . . . 38
7.12 Soft-State Handling . . . 38
7.13 Simplifying Assumptions and Decisions . . . 38

8 Licensing **40**

9 Testing **41**

10 Fulfilment of Requirements **59**

11 Experimental Scenario Applications **61**
11.1 IPv6 emulation . . . 61
11.2 Common Experimental Features . . . 62
11.2.1 Topology Setup . . . 62
11.2.2 Simulation of Delay . . . 64
11.2.3 Emulation Configuration . . . 65
11.2.4 Reporting of Emulated IoT Devices' Statistics . . . 66
11.2.5 Communication Between Application Objects and the Emulated IoT Devices . . . 67
11.2.6 Experimental Assumptions . . . 67
11.2.7 Statistical Handling . . . 67
11.3 Node-Centric Temperature Sensor Network . . . 69
11.3.1 Description . . . 69
11.3.2 Configuration . . . 70
11.3.3 Implementation . . . 71
11.3.4 Method . . . 72
11.3.5 Results . . . 73
11.4 Data-Centric Temperature Sensor Network . . . 83
11.4.1 Description . . . 83
11.4.2 Configuration . . . 83
11.4.3 Implementation . . . 83
11.4.4 Method . . . 84
11.4.5 Results . . . 84
11.5 Animal Tracker . . . 92
11.5.1 Description . . . 92
11.5.2 Configuration . . . 94
11.5.3 Implementation . . . 95
11.5.4 Method . . . 96
11.5.5 Results . . . 97
11.6 Emergency Communication Network . . . 102
11.6.1 Description . . . 102
11.6.2 Configuration . . . 103
11.6.3 Implementation . . . 104

11.6.4 Method 105
11.6.5 Results 106

12 Evaluation and Conclusions **111**
12.1 Experimental Conclusion 111
12.2 Summary 112
12.3 Objectives Completed 112
12.4 Critical Appraisal 113
12.5 Future Work 114

References **118**

13 Appendices **119**
13.1 User Manual 119
13.1.1 API 122
13.1.2 Running experiments 122
13.2 Emulation API Pydoc 122
13.3 Updated Semantics for Locator Updates 126
13.4 Preliminary Ethics Self-Assessment Form 139
13.5 Description, Objectives and Ethics and Resources Form 143
13.6 Remaining Graphs 146
13.6.1 Nodecentric TSN 146
13.6.2 Datacentric TSN 148
13.6.3 Animal Tracker 150
13.6.4 Emergency Communication Network 156

3 Introduction

Weiser's vision of ubiquitous computing clearly provides great utility to the user, as it emphasises the need for greater self-configurability to minimise user intervention, while providing more computing power. Recent innovations in IoT technology, such as their applications in healthcare which allow remote monitoring of patients [34] appear to meet this vision in providing wider access to computing power.

However to fully implement ubiquitous computing, IoT devices must also perform ubiquitous communication. Nodes which implement ubiquitous communication are able to distribute any transport layer flow over multiple connectivities, and maintain a flow while its connectivity is changing[41]. This provides path redundancy, which widens access to computing power by providing more resilient connectivity, and seamless mobility which means the user can move where they wish without experiencing a disrupted flow, both of which facilitate Weiser's vision. Unfortunately, ubiquitous communication is not possible over an IP stack, and therefore IoT devices cannot fully facilitate Weiser's vision.

It is the overloading of IP addresses as identifiers and topological locators that makes IP unsuitable for ubiquitous communication. Back when IP was being developed, computers were larger and more stationary. Therefore network topology was less dynamic, and this overloading made sense. However, the advent of smaller wireless mobile devices (including IoT devices) resulted in more dynamic network topologies, and the assumption that an IP device would remain in the same topological location no longer held.

This presented a problem: a new IP address is required to take advantage of connectivity from a different topological location. However, the old IP address would be used as the identifier for any existing transport level flows as figure 1 shows. Therefore when changing connectivity, the existing transport layer flows are broken, and new ones are created with the new address, and informing the correspondent nodes of this change will incur control plane overhead[40]. Additionally, distributing a flow over multiple connectivities would not be possible for a single transport layer flow as multiple connectivities would require multiple addresses and, therefore, multiple transport layer connections.

Table 1: Use of names in IP and ILNP.

Protocol layer	IPv4 and IPv6	ILNP (ILNPv6)
Application	FQDN, IP address	FQDN or app.-specific
Transport	IP address	Node Identifier (*NID*)
Network	IP address	Locator (*L64*)
(interface)	IP address	dynamic binding

Figure 1: Table showing the names used at different layers under IP and ILNP. [36]

An alternative to IP is the Identifier-Locator Network Protocol, which dispenses with the overloaded IP address entirely and instead adopts separate identifier and locator values, which represent the node's identity and topological location respectively. The topologically significant locator value is only used as an identifier at the network layer, and the node identifier is used as the transport layer identifier, as figure 1 shows. This means any changes in connectivity do not invalidate existing transport layer flows and that multiple locators can be used to distribute a single transport layer flow over multiple connectivities, thus allowing ILNP-enabled nodes to perform ubiquitous communication.

The aim of this project is to evaluate the relative suitability of ILNP-enabled IoT devices for ubiquitous communication in comparison to IPv6-enabled IoT devices. To this end, the following objectives were devised.

3.1 Objectives

These objectives were initially specified in the Description, Objectives, Ethics and Resources document, which can be found in the Appendices section of this report.

3.1.1 Primary Objectives

P1 Design and implement a library in Python which emulates IoT devices using an overlay network. These emulated IoT devices will use an Identifier-Locator communication protocol over IPv6.

P2 Design and implement an API to allow users to build applications for this emulated IoT platform.

P3 Provide an interface for control and management of the overlay network.

3.1.2 Secondary Objectives

S1 This library should be packaged and licensed so third parties may use it.

S2 This library should provide a mechanism to measure resource usage.

3.2 Project Overview

Firstly, a context review of existing work on ubiquitous systems, ILNP and IoT technology is presented. Then the objectives are decomposed into a requirements specification, and the design and implementation of the overlay network is discussed. Following this, the testing of the emulation is presented.

Comprehensively meeting all of the defined objectives made the implementation of a series of experimental scenario applications possible, and their design and implementation are presented next part of this project. These scenarios make use of an increasing amount of ubiquitous communication in the form of mobility and multipath communication, and they are as follows:

- A temperature sensor network which uses a nodecentric paradigm[35]. This makes use of multipath communication.
- A temperature sensor network which uses a datacentric paradigm[35]. This makes use of multipath communication.
- An animal tracking application which makes use of multipath communication and the mobility of a single node which represents the IoT device attached to the animal.
- An emergency communication network which makes use of multipath communication and the mobility of every node.

Finally, the results of these experimental scenario applications run over the ILNP emulation and a similar IPv6 emulation are presented and compared to determine the relative suitability of ILNP-enabled IoT devices for implementing ubiquitous computing.

Note that this project presents an emulation of ILNP, not a full implementation, so some features of ILNP which are outside of the scope of this project are omitted.

4 Context Review

4.1 Ubiquitous Systems

Ubiquitous systems were first proposed by Mark Weiser in his 1993 paper "Some computer science issues in ubiquitous computing"[40]. He starts by defining ubiquitous computing as a paradigm that makes computing power "available through the physical environment" and emphasises that these computers must operate so effectively and with minimal need for configuration that they are "effectively invisible"[40].

To contrast ubiquitous computing to the state of computing at the time of writing, he points out that while computing which is locally accessible can perform useful tasks, the computer becomes the "focus of attention"[40] instead of an aid to the human that goes unnoticed while they perform the task as they normally would, which would be preferable as the user would not need to expend effort getting the computer to perform as intended.

He then describes hardware innovations which partially satisfy the definition of ubiquitous computing and emphasises the need for low power, wireless devices and the utility of pen-like devices for meeting his vision.[40]

Next, he states that medium access control protocols used in ubiquitous systems must allow for mobile nodes frequently joining and leaving networks while preventing any given node from being starved of connectivity, and allow for real-time communication[40]. He also states that medium access control mechanisms for wired networks are insufficient for wireless networks because of the "hidden terminal" problem in wireless networks, therefore, wired MAC algorithms cannot be used to implement ubiquitous computing. Weiser describes modifications to a protocol called MACA which resolves these issues. The shortcomings of IP described in the introduction are pointed out by Weiser too.[40]

Weiser also raises concerns about location privacy issues with ubiquitous systems, as some form of location data is required to forward packets to a node. He discusses centralised and distributed approaches to storing this data, pointing out that while a centralised repository could be well protected, a single security incident there would very costly, and a distributed approach would allow users to customise the security features they have to protect their location data[40].

He also discussed methods that ubiquitous systems could use to allow users to interact with them, and applications for ubiquitous systems which emphasise the utility of his vision[40].

4.2 ILNP

RFC6740[21] details the design of ILNP. It describes how the IP address is abandoned in favour of the Identifier-Locator Vector(I-LV) which consists of a node identifier, and a topological locator. In contrast to IP which uses the IP address as the identifier at both the network layer and the transport layer, ILNP uses node identifiers as the transport layer identifier, and locators as the network layer identifier. A node can have multiple identifiers, each with a dynamic mapping to a set of locators. By manipulating the set of locators an identifier maps to, a single transport flow identified by this node identifier can persist under mobile conditions and use multiple connectivity while maintaining a consistent end-to-end transport state[21].

RFC6740 also mentions some characteristics of ILNP which make deployment easier. It emphasises the potential to adapt existing IP mechanisms to enable basic connectivity, such as DHCP and DNS with the extra Resource Record types outlined in RFC6742[19]. Additionally, it mentions that ILNPv6 (which is what this project emulates) is designed as a superset of IPv6, which means that their addresses look the same on the wire, and IPv6 control messages such as router advertisements can be used by ILNPv6 too. Finally, the fact that ILNP locators are semantically equivalent to IPv6 routing prefixes means that IPv6 routers can forward ILNP packets with no changes[41]. This means that the deployment of ILNP can be entirely done on end systems which will make deployment easier and hopefully increase uptake.

In addition to the existing IP control mechanisms that can be adapted for use by ILNP, ILNP also uses the Locator Update mechanism[17]. This control message exists to inform correspondent nodes which have recently communicated with a mobile node of any changes in the locator set that it is associated with. The semantics outlined in RFC6743[17] stipulate that locator updates contain every valid locator of the mobile node, however, an update to the semantics is being considered in which the reserved field will be changed to indicate whether the locators in the message are just the new ones added to the set, the ones which should be removed from the set, or just the entire set. These updated semantics are not publically available yet, however, they are available in the "Updated Semantics for Locator Updates" appendix. Additionally, a bit in the reserved field could also be used to indicate whether soft handoff or hard handoff should be used. When mobile nodes use soft handoff, an acknowledgement from all correspondent nodes must be received before finalising the changes to the locator set, whereas they would not have to wait for any acknowledgements if hard handoff was used.

While this project mainly deals with ILNP's native support for host-level multipath communication and mobility, the ILNP homepage [24] describes many more benefits of using ILNP. These include: native support for IPSec and other applications which rely on a consistent end-to-end transport state, as the re-

moval of topological information from the transport level identifier means that a consistent end-to-end transport state can be maintained in the face of changing connectivity; the potential for networking APIs which better support multiple connectivity, and native support for site and host multihoming[24].

ILNP's support for multipath communication is equivalent to host-level multihoming[20]. However, ILNP also supports site-level multihoming as nodes may use their identifier with the locator associated with any of its upstream providers to form a valid Identifier-Locator vector, without impacting any existing transport flows as locators do not appear in the transport state. This also enables ILNP site-multihoming to use provider aggregated addressing while still providing resilient transport layer flows in the face of an uplink failure, as the node can use a different locator for the same transport layer connection. If IPv6 wishes to provide this level of resilience in the face of an uplink failure while multihoming, it must use provider-independent addressing, which does not scale well and bloats the global routing state[21]. Therefore unlike IPv6, ILNP's multihoming mechanism would not grow the global routing state at an unsustainable rate.

This project provides a userspace emulation of ILNP-enabled IoT devices, however, the ILNP homepage also provides a link to a Linux kernel implementation of ILNP. This kernel implementation was used to perform experiments which test the gratuitous packet loss incurred when using ILNP's soft handoff mechanism[36]. The results showed that the usage of ILNP's soft handoff mechanism resulted in nearly no gratuitous packet loss because the mobile node must receive an acknowledgement before the locator it is leaving stops being usable, at which point the correspondent node is guaranteed to know the new locator being used.

Finally, like IPv6, ILNP can provide identity privacy by generating different identifiers for each of its transport layer sessions, which means it would be more difficult to correlate which flows belong to the same device. However, in contrast to IPv6, ILNP's support for location privacy through the use of Locator Rewriting Routers will maintain a consistent end-to-end transport state as the locators are not part of the end-to-end transport state, whereas attempts to perform address translation to enforce location privacy in IP will make the end-to-end transport state inconsistent[41].

4.3 IoT

The paper entitled "Communication Paradigms for Sensor Networks"[35] details some addressing paradigms which IoT sensor networks can use. It describes the traditional nodecentric approach in which sensors use the addresses of a sink to publish their readings, however a datacentric approach is also presented in which sinks broadcast their interest in certain types of data and the sensors send their data in the direction it received expressions of interest in that data. Finally,

a position centric approach is presented in which nodes use their knowledge of the physical location of the destination node and send packets to a next hop in the physical direction of this destination node. In this project, two experimental scenario applications will be set up which demonstrate the nodecentric and datacentric approaches to temperature sensor networks.

The paper entitled "Modeling and Comparison of Delay and Energy Cost of IoT Data Transfers"[33] presents measurements of the delays incurred and energy consumed by different IoT technologies when sending different amounts of data over different ranges. The delay data was used to inform the chosen artificially induced delay of 10ms in the experiments, and the energy values could also be used to estimate the energy consumption of the emulated IoT devices in future work.

5 Requirements Specification

The objectives of this project were decomposed into the following set of specific requirements. The extent to which this project satisfies the objectives can be determined by how well it fulfils this set of requirements.

5.1 Functional Requirements

5.1.1 Functional Requirements of Objective P1

F1 Identifier-Locator mappings of remote nodes should be advertised and learned through the flooding of neighbour advertisements.

F2 ILNP's propensity for multipath communication should be exploited through a deficit round robin algorithm which chooses among a set of valid remote locators when sending to a remote identifier.

F3 A node should be able to demand connectivity information through the transmission of neighbour solicitations and router solicitations.

F4 ILNP's propensity for mobility should be exploited through the implementation of locator updates. Any node which has recently engaged in unicast network layer communication with the mobile node should be classed as a correspondent of the mobile node, and should receive a locator update when the mobile node performs mobility.

F5 ILNP nodes which provide routing services should advertise their presence in the overlay network with regular router advertisements in a locator accessible to all nodes. These advertisements should contain each invariant locator they can provide access to.

F6 Forwarding tables should be built up through the interpretations of neighbour advertisements.

F7 Soft-state should be used to gracefully degrade in the face of the following state becoming stale:

- Active correspondents
- Identifier-Locator mappings of remote nodes
- The locators the node is associated with
- Available routers
- Forwarding table

F8 Identifier and Locator values should be automatically generated using the MAC address of the machine the emulated node is running on.

F9 The network identified by each locator should map to a distinct IPv6 multicast group.

F10 An emulated transport layer with source and destination ports should be provided to allow each emulated IoT to multiplex and demultiplex several transport layer flows simultaneously.

F11 An emulated link layer should be provided to allow nodes to uniquely specify the next node in a path, and register the previous node in a path.

F12 Transport level flooding loops should be prevented with the use of unique flood numbers maintained by each node.

F13 The code should make use of asynchronous programming to provide highly responsive code without the overhead of a busy wait or multiple threads or processes.

5.1.2 Functional Requirements of Objective P2

F14 API calls should be serviced by the emulated IoT nodes over an IPv4 control channel.

F15 The API too should make use of asynchronous programming to allow applications to be event-driven and prevent blocking while waiting on network events to occur.

F16 The API should provide appropriate return values to indicate the success or failure of an operation.

5.1.3 Functional Requirements of Objective P3

F17 The control interface should be implemented in API calls.

F18 The management interface should be implemented as a server which can be run as a systemd service or as a TMUX session. This management server should serve any API requests to start an emulated IoT on the machine on which the server is running.

F19 The API should also contain a management primitive to shut down an emulated IoT and return the remote machine to a state where the server is waiting for an API call to start an emulated IoT on that machine.

F20 The management server should also serve handshake API calls and return whether it is available to start an emulated IoT. The API call initiating this handshake should be configured with a timeout in case the management server is not running.

5.1.4 Functional Requirements of Objective S1

F21 The emulation should be packaged with a requirements.txt file to allow the user to install any required dependencies easily.

5.1.5 Functional Requirements of Objective S2

F22 CPU and memory usage should be recorded using the "psutil"[30] Python library.

F23 The API should provide functions to request the CPU usage, RSS (a measure of physical memory) and virtual memory used by the process running an emulated IoT.

5.2 Non-Functional Requirements

5.2.1 Non-Functional Requirements of Objective P1

NF1 The ILNP emulation should use addressing that employs an identifier-locator split, which will permit it to emulate ILNP's native support for mobility and multipath communication

NF2 Emulated IoTs should self-configure as much as possible, in order to minimise the necessary user intervention.

NF3 The emulated overlay network should be an ad-hoc network.

NF4 The emulated IoTs should perform forwarding and routing.

NF5 The emulation program should be robust and be able to handle the exchange of high volumes of data in mobile and multipath scenarios.

NF6 The emulation should be able to scale to support applications involving the active participation of at least ten nodes.

NF7 The emulation must be able to run on a network of Python-enabled Linux machines which are on the same link-local IPv6 network. They must also be IPv4 enabled.

NF8 Configurable parameters of the each emulated IoT devices should be defined in a YAML configuration file to allow the user to customise the behaviour of the emulation.

5.2.2 Non-Functional Requirements of Objective P2

NF9 The API should allow the user to instruct emulated IoTs to perform the following in order to create an application:

- Unicast sending and receiving of application data
- Flooding of application data.
- Opening and closing transport level sockets

NF10 The API should provide getters which allow the applications to obtain information about the state of the emulated IoT.

NF11 The API should allow the user to write event-driven code which avoids blocking when a task must wait for an event.

5.2.3 Non-Functional Requirements of Objective P3

NF12 The control interface should allow the user to instruct an emulated IoT to perform the following:

- Join locators.
- Leave locators.
- Replace the entire locator set.

NF13 The management interface should be able to set up a topology defined in a passed YAML file.

NF14 The management interface should allow the user to start and stop an emulated IoT at will on a remote machine that is running the management interface server.

NF15 The management interface should be able to take in a list of hostnames, and determine which of the passed hosts actually are available to host a new emulated IoT.

NF16 The management interface should support the management of at least ten emulated IoTs with minimal user intervention.

5.2.4 Non-Functional Requirements of Objective S1

NF17 The emulation should be packaged with open-source licensing.

NF18 The emulation should be packaged with a User Manual, which contains clear instructions on how to set up and run it, and a README that contains a code manifest and licensing information.

5.2.5 Non-Functional Requirements of Objective S2

NF19 The emulation should provide a mechanism to observe the CPU usage, RSS (a measure of physical memory) and virtual memory used by the process running the emulation.

6 Ethics

There are no ethical concerns associated with this project, as the signed preliminary ethics self-assessment form in the "Preliminary Ethics Self-Assessment Form" appendix shows.

7 Emulation Design and Implementation

7.1 Overlay network

This emulation is implemented as an overlay network in which each emulated IoT will run on a separate computer within the same IPv6 network. A discussion of each layer of this overlay network follows:

7.1.1 Network Layer

To implement the topological partitions in the overlay network between the subnetworks represented by different locator values, IPv6 multicast was used. A 64-bit IPv6 link-local multicast prefix is defined in the configuration file, and the 64-bit Locator values are appended to this to form an IPv6 multicast address which will be joined by the underlying machines that host any emulated IoTs that associate with this locator.

There is a broadcast locator `FF:FF:FF:FF` which all devices join upon booting up, and its sole purpose is for the transmission of Router Advertisements so every device can see every advertised locator. While this does mean that every emulated IoT will receive every router advertisement which would not happen in a wireless IoT network due to range constraints, this is necessary to provide basic connectivity without requiring the user to hardcode the locators each emulated IoT should join. Additionally, the user may encode these range constraints into the applications they build if they so wish.

There is also a broadcast identifier `FF:FF:FF:FF` that indicates the ILNP datagram is intended for every node that received it. This is used when router advertisements are sent to the broadcast locator, and for any advertisements or application data that undergoes a controlled flooding.

The ILNP header is as follows:

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Next Header	Is Router	Unused
Unused		
Source Identifier		
Source Identifier Continued		
Source Locator		
Source Locator Continued		
Destination Identifier		
Destination Identifier Continued		
Destination Locator		
Destination Locator Continued		
Payload		
...		

Figure 2: ILNP Header Packet Diagram

A list of the semantics of the fields in the ILNP header follows:

- Next header: Whether the payload is a control message or transport datagram
- Is router: Indicates whether the source node performs routing and forwarding.
- Source Identifier: Identifier of the source node
- Source Locator: Locator the source node used to transmit this packet
- Destination Identifier: Identifier of the destination node
- Destination Locator: Topological location the packet is being sent to

The ILNP header this project uses differs from the ILNP header that is identical to the IPv6 header apart from the changed semantics of the address bits as this emulation only seeks to emulate a subset of the functionality of ILNP, so some fields like "Traffic Class" were deemed unnecessary and omitted.

7.1.2 Transport Layer

It is necessary for emulated IoT devices to be able to engage in several transport layer flows simultaneously. Therefore a bare-minimum transport layer is

implemented that uses port numbers to multiplex and demultiplex flows.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Source Port
Destination port
Flood Number
Payload
...

Figure 3: Transport Header Packet Diagram

Transport layer headers may be sent to every reachable node by performing a controlled flooding in which nodes send the packet on every locator except the one it received it on using the broadcast identifier. The flood number forms a tuple `(Source Identifier, Flood Number)`, which acts as a unique identifier for each datagram which is being flooded. Recording which packets have already been flooded by a particular node is done to prevent routing loops.

There is a configuration parameter called `transport_mss` which defines how many application bytes can be sent per transport datagram before the emulation fragments it into multiple transport datagrams. It is important to set this so that the underlying network does not perform fragmentation in the middle of a single transport datagram.

7.1.3 Link Layer

When the emulated IoT receives a packet, it must be able to determine whether the datagram is intended for it or for another emulated IoT associated with the destination locator. Therefore a bare-minimum link layer is implemented. Any link layer headers whose destination identifier does not match the recipient's identifier or the broadcast identifier are dropped.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Source Identifier
Source Identifier Continued
Destination Identifier
Destination Identifier Continued
Locator
Locator Continued
Payload
...

Figure 4: Link Frame Packet Diagram

Note that since the source and destination locators will always be the same for a single hop link layer transmission, the link layer header only includes a single locator.

7.1.4 Overall Overlay Network Stack

Combining the above layers results in the overall stack for the emulation's overlay network as shown below:

Application
Custom Transport Protocol
ILNP Emulation
Custom Link Layer Protocol
UDP
IPv6
Link Layer Ethernet

Figure 5: Diagram Showing the Overlay Network Stack

7.2 Router Advertisements

Not every emulated IoT will take part in routing and forwarding, so those that do advertise this by intermittently broadcasting router advertisements. The advertisements contain the invariant locators the router owns, and this indicates to the receiving nodes that they have connectivity to the network through this locator.

The reasoning for a set of locators that each node exclusively owns called "invariant locators" in an ad-hoc ILNP network is that a locator should represent a

specific topological location, however, if two devices that have joined the same locator lose contact, the situation arises where each node advertises they are available on this locator even though they no longer share a topological location. To solve this, an owner node will have it as an invariant locator, which means it always has it as a locator and other nodes' right to associate with that locator is conditional on having recently received a router advertisement advertising this locator from the owner. This means contact has not been lost with the owner that defines the topological location for that locator. Therefore, the other node can honestly advertise that it is available via that locator.

Each router sends its router advertisements intermittently onto the broadcast locator where every node can see it. If a node sees a locator in a router advertisement, it will be cached in `available_routers`.

Nodes can choose to join the invariant locators of any routing IoT in `available_routers`. Conversely, if a node has joined a locator and has not seen it in a router advertisement for a configured timeout, the locator is left without using soft handoff, as soft handoff is impossible as contact has been lost with the router, so other nodes will receive an updated version of the locator set through the next neighbour advertisement.

Figure 6 shows the packet diagram for the router advertisement.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Type	Number of Locators	Unused
Router advertisement number		
Identifier		
Identifier continued		
Locator [1]		
Locator [1] continued		
...		

Figure 6: Router Advertisement Packet Diagram

The router advertisement number is used to chronologically order routers. Nodes keep track of the latest router advertisement number seen from each router to avoid processing old ones.

Finally when an IoT starts up it will need to quickly request connectivity information from the routers available to it, and this is achieved by sending a router solicitation. This is sent on the broadcast locator and any router that sees it will respond with its router advertisement on the broadcast locator.

Figure 7 shows the packet structure of the router solicitation.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Type								Router Solicitation Number							

Figure 7: Router Solicitation Packet Diagram

The router solicitation number is used to keep track of the most recent router solicitation from each node, and avoid responding to old router solicitations.

7.3 Soft Vertical Handoff

Soft handoff is achieved by mobile nodes keeping track of correspondent nodes, and when it changes its locator set using soft handoff it will send a Locator Update message to all these correspondent nodes to inform them of this change.

Correspondent nodes are defined as those which have recently engaged in unicast communication with the mobile node, as this means that they are therefore likely to attempt unicast communication with the mobile node in the future and will therefore benefit from being informed of any change in connectivity. The reason nodes which engage in broadcast communication with the mobile node are not counted as correspondents is that every node engages in broadcast flooding with every other node, and also broadcast communication will work no matter what connectivity the mobile node has, so being informed of changes in connectivity will not improve the effectiveness of broadcast communication.

The mobile node waits for an acknowledgement from all its active correspondents (in the form of an explicit ACK, or the reception of a packet on one of the new locators) before it stops receiving on the locators it is leaving. If a correspondent does not reply, the locator update is retransmitted 5 times until the change in locator set is enacted anyway. If a correspondent times out, the requirement that an acknowledgement must be received from it in order to enact an outstanding locator update is waived.

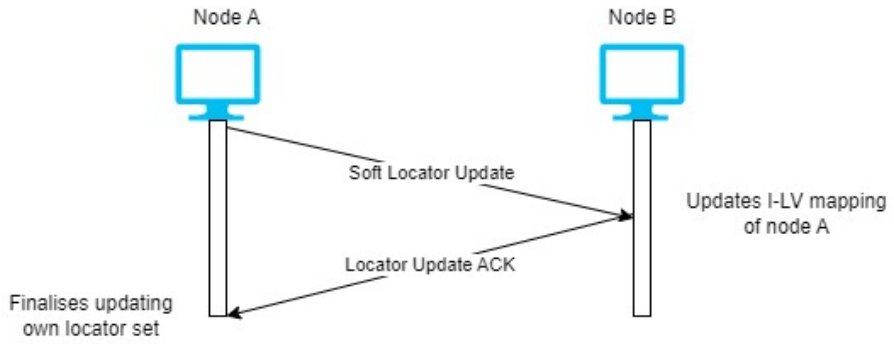

Figure 8: Soft handoff sequence diagram

Locator Update messages are a slightly simplified version of their specification in RFC6743[17], and they make use of the updated semantics outlined in the document contained in the "Updated Semantics for Locator Updates" appendix provided by Saleem Bhatti, which is not public yet.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Type	Action	Number of locators	Operation
Locator set number			
Identifier			
Identifier continued			
Locator [1]			
Locator [1] continued			
...			

Figure 9: Locator Update Packet Diagram

Here follows a brief explanation of the semantics of each field.

- Type: Indicates that this datagram is a Locator Update.
- Action: Indicates the type of Location Update that it is dealing with, whether it is adding, deleting or replacing an existing locator set, and whether this is done with the soft handoff mechanism or not.
- Number of locators: Number of locators associated with this Locator Update
- Operation: Whether this is the initiation of a Location Update, or just an acknowledgement.

- Locator set number: Indicates the chronological ordering of this locator set for this node. Enforces consistency of I-LV mappings across the emulation as the order of locator updates and neighbour advertisements matter. Old locator sets are ignored.
- Identifier: Identifier for the node whose locator set is being updated.
- Locator(s): Locators which are being added to, deleted from or replacing the previous locator set.

All the API calls which modify locator sets use soft handoff, however there is another mechanism called hard handoff in which the mobile node does not wait for any acknowledgement to start using the new locator set. This is only used when a router advertisement for a locator has not been received in a while, so it times out and the node abandons using the locator. In this case, a neighbour advertisement is used to advertise the updated connectivity instead of a hard locator update, which is an implementation convenience that has the same effect.

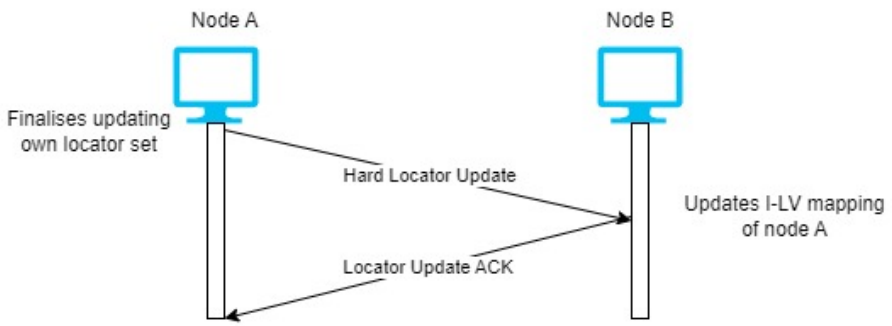

Figure 10: Hard handoff sequence diagram

To make the soft handoff mechanism more robust a node may only have one outstanding locator update at a time. Any additional attempts to change the locator set are denied until the current locator update is resolved. This includes when locators time out when a different locator update is pending, in which case the timed out locators are cached, and will be left using hard handoff immediately after the current locator update is resolved.

7.4 Handling of Own Locators

For a node to associate with a locator that is not one of its invariant locators, it has to have received a router advertisement advertising it recently, and the locator will time out if no advertisement is received for a while. Additionally, the state of a locator for a node goes through some transitional states when soft

handoff is being performed. Emulated IoT's keep track of the state of their own locators using flags which are loosely based off those specified in RFC6741[22]. They are as follows:

- `ACTIVE`: The locator is in active use.
- `AGED`: The locator was in active use. It has timed out, but has not yet been deleted and may come back to being in active use. May not participate in flows.
- `VALID_JOIN`: The locator is a subject of a soft join locator update. May participate in flows.
- `VALID_LEAVE` The locator is a subject of a soft leave locator update. May participate in flows.

A finite state machine indicating how flag state changes across the lifetime of the emulated IoT is shown below:

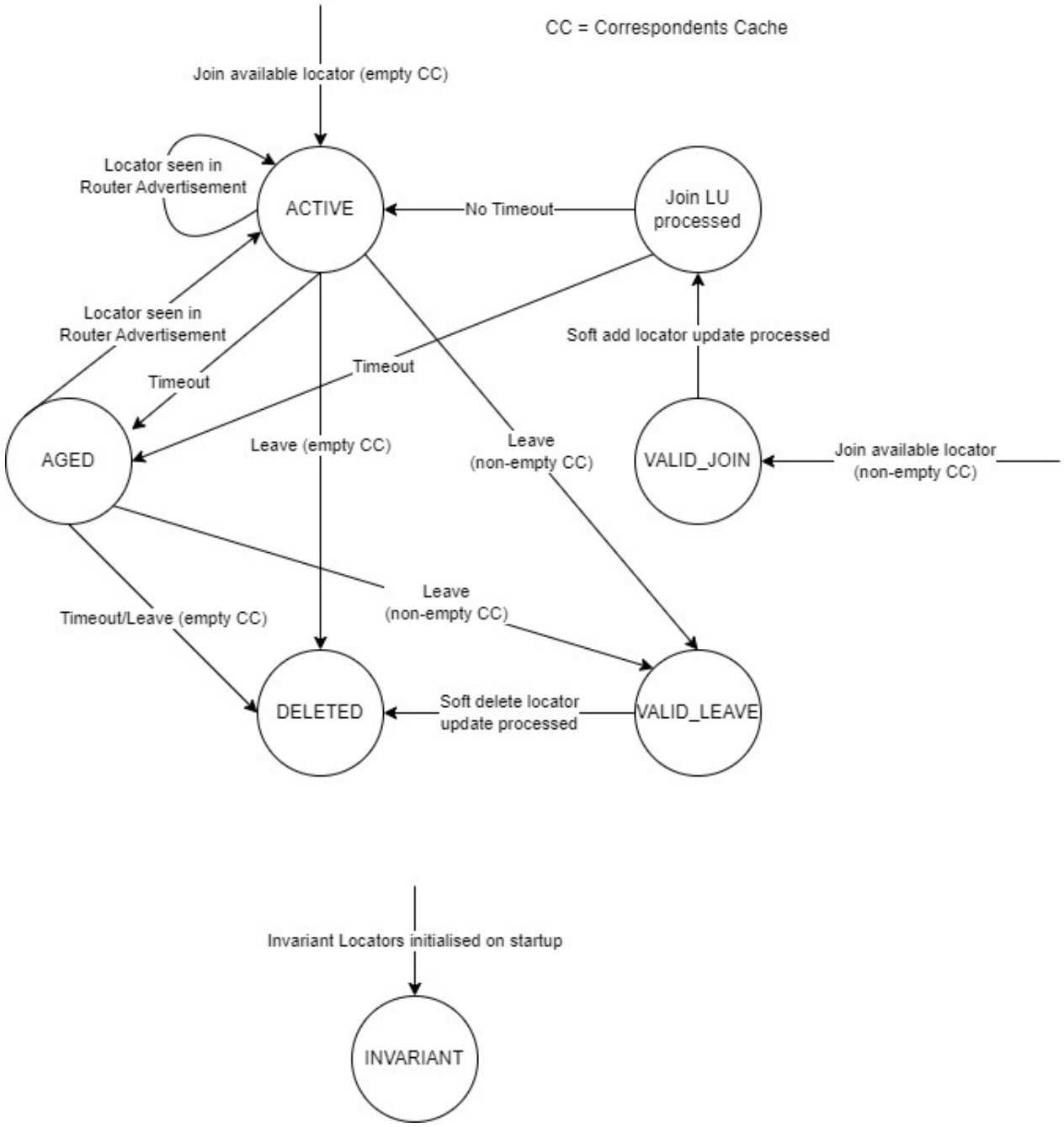

Figure 11: FSM for how flag state changes for the emulated IoT's own locators.

Note that invariant locators are not stored with the rest of the locators and do not have a flag, but are included in the FSM for the sake of completeness.

7.5 Neighbour Advertisements

In order for nodes to become aware of each other, some sort of discovery protocol must be implemented. In infrastructure networks, DNS is usually implemented. However this emulation deals with a completely ad-hoc network so DNS was not used.

Instead, a flood and backward learning approach was used in which every node intermittently floods a neighbour advertisement onto the network. The structure of this neighbour advertisement is shown in figure 12.

<table>
<tr><th colspan="8">0 1 2 3 4 5 6 7</th><th colspan="8">8 9 10 11 12 13 14 15</th><th colspan="16">16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31</th></tr>
<tr><td colspan="8">Type</td><td colspan="8">Is Router</td><td colspan="16">Number of Locators</td></tr>
<tr><td colspan="16">Number of Hops</td><td colspan="16">Unused</td></tr>
<tr><td colspan="32">Neighbour Advertisement Number</td></tr>
<tr><td colspan="32">Locator Set Number</td></tr>
<tr><td colspan="32">Identifier</td></tr>
<tr><td colspan="32">Identifier continued</td></tr>
<tr><td colspan="32">Locator [1]</td></tr>
<tr><td colspan="32">Locator [1] continued</td></tr>
<tr><td colspan="32">...</td></tr>
</table>

Figure 12: Neighbour Advertisement Packet Diagram

Below are the semantics of each field in the Neighbour Advertisement:

- Type: Indicates it is a Neighbour Advertisement
- Is Router: Indicates whether this neighbour takes part in routing and forwarding
- Number of Locators: The number of Locators this node is advertising that it is available on
- Number of Hops: The number of hops this advertisement travelled to get to the receiving node.
- Neighbour Advertisement Number: Chronologically orders neighbour advertisements from a node. Receiving nodes use this to ignore old neighbour advertisements.

- Locator Set Number: Chronologically orders the locator set being advertised. Locator Updates use this number for ordering too. Receiving nodes use this to determine which locator set they received is the most up to date.
- Locator(s): Locators the source IoT is reachable on.

Upon reception of a neighbour advertisement, the node updates two bits of state: the forwarding table, and the I-LV cache.

The forwarding table records which locators are accessible from which next hop routers, and the reception of a neighbour advertisement from a non-routing node indicates that the source locator (accessible from the ILNP header) is accessible from the router which forwarded the neighbour advertisement to it (accessible from the link frame header). Additionally, if the node which initially sent the neighbour advertisement performs routing and forwarding (as indicated in the advertisement) then every locator it is associated with is accessible from the router which forwarded the neighbour advertisement to the recipient, so all of those paths are recorded too.

The number of hops is also recorded in the forwarding table as when sending to a destination locator, the entry with the least number of hops is chosen. When forwarding a neighbour advertisement the number of hops is incremented.

The I-LV cache is the record each node has of which locators each other node in the emulation this is associated with. This record is formed by the processing of Neighbour Advertisements and Locator Updates. Upon reception of a neighbour advertisement, the node checks it is advertising a current locator set by checking the `Locator Set Number` and if it is old, it is discarded. Otherwise, the I-LV cache entry for the sending node is updated to the locators in the advertisement. Locators are stored with flags which indicate their status like in RFC6741[22]. The flags are:

- `ACTIVE`: The locator is in active use.
- `EXPIRED`: The locator has timed out, however data has been received from it which indicates it is still in use, so it may still be used to exchange flows.
- `AGED`: The locator was in active use. It has timed out, but has not yet been deleted and may come back to being in active use. Will not be used to send data to, however receiving data from it will make it `EXPIRED`.

Below is a diagram which shows how the flags for a locator in the I-LV will change throughout the liftime of the emulation.

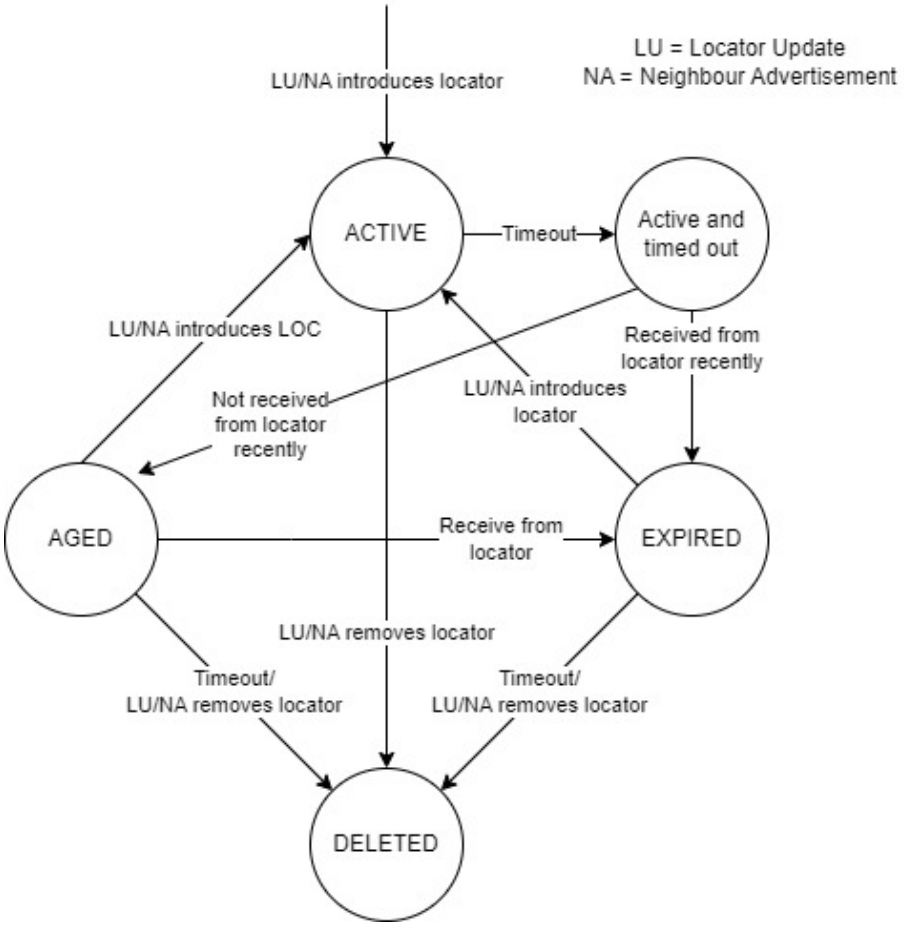

Figure 13: FSM for how flag state changes for the other emulated devices' locators.

As shown above, once an up-to-date neighbour advertisement or locator update introduces an I-LV mapping it is marked as active. Once it times out, it can be considered `EXPIRED` if it has been used recently or `AGED` if it has not. This distinction is present because if other packets from this I-LV mapping have been received, that is an indicator that it is still available, so recording this distinction is important. Finally, if an up-to-date neighbour advertisement or locator update removes a locator, or another timeout occurs, it is deleted.

Finally, emulated IoTs can send neighbour solicitations in order to obtain connectivity information instantly when it starts up. The packet structure of the neighbour solicitation is shown in figure 14.

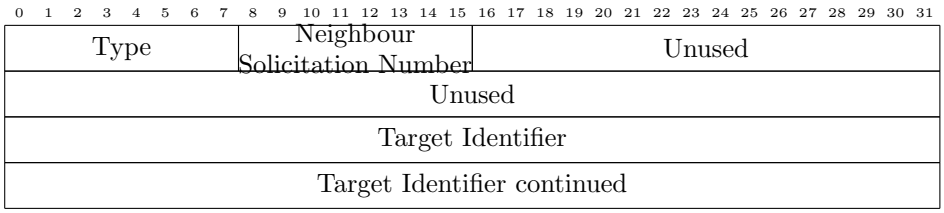

Figure 14: Neighbour Solictation Packet Diagram

Upon receiving a neighbour solicitation, the node uses the neighbour solicitation number to check if it is a new neighbour solicitation, and if the target identifier is either the node's identifier or the broadcast identifier, the recipient floods its neighbour advertisement through the network.

7.6 Sending

A key primitive that this emulation provides is unicast communication between ILNP nodes. This is done by using set operations to determine which of the destination node's locators are reachable, and which next hop routers can be used. The pseudocode is outlined below.

Algorithm 1 Sending algorithm

$remote_locs \leftarrow$ `ACTIVE` and `EXPIRED` locs in $ilv_cache[dst_id]$

$own_usable_locs \leftarrow$ source node's locs which are not`AGED`

$available_next_hops \leftarrow$ routers which are reachable on own_usable_locs

$forwardable_dst_locs \leftarrow$ locators in $forwarding_table$, whose next hop is in $available_next_hops$

$reachable_dst_locs = (forwardable_dst_locs \cup usable_src_locs) \cap remote_locs$

$chosen_dst_loc = DRR(dst_id, reachable_dst_locs, payload_len)$

if $chosen_dst_loc$ in own_usable_locs **then**
  $link_dst_id \leftarrow dst_id$
  $link_dst_loc \leftarrow chosen_dst_loc$
else
  $potential_next_hops \leftarrow forwarding_table[chosen_dst_loc] \cup available_next_hops$
  $link_dst_id \leftarrow potential_next_hops[0]$
  $potential_next_locs \leftarrow$ `ACTIVE` and `EXPIRED` locs from $ilv_cache[link_dst_id]$
  $link_dst_loc \leftarrow potential_next_locs[0]$
end if
Send using link packet with destination ID $link_dst_id$ and locator $link_dst_loc$

ILNP's capacity for multipath communication is part of the reason it is very suitable for ubiquitous communication. This multipath communication is enabled by using a deficit round robin algorithm [23] to choose between several valid destination locators for a flow. The deficit round robin algorithm effectively keeps track of how much each I-LV was passed over, and each time it chooses the I-LV that has been passed over the most, which results in equitable load distribution. The pseudocode is shown below.

Algorithm 2 Deficit Round Robin Algorithm

```
Initialise deficit counters if necessary
for i = {0..payload_len/QUANTUM} do
    for loc in reachable_dst_locs do
        c = QUANTUM + deficits[(dst_id, loc)]
        if payload_len ≤ c then
            deficits[(dst_id, loc)] = c − len
            return loc
        end if
        deficits[(dst_id, loc)] += QUANTUM
    end for
end for
```

As an aside, forwarding is performed using the same mechanism as sending, except the destination locator has already been decided by the source node.

7.7 Generation of Node Identifiers and Locator Values

To make usage of this emulation easier, identifiers and locators are automatically generated. This is done by obtaining the 48-bit MAC address of the underlying machine on which the emulated IoT device is running, and adding 16 bits onto it to produce a 64-bit identifier or locator.

Every emulated IoT uses just a single identifier so the bits added onto the MAC address to obtain the identifier are just 0, however if the emulated IoT device has invariant locators, the bits added onto the MAC address to obtain these locators are the numbers from 0 to the number of invariant locators required. This mechanism will allow each device to have 2^{16} invariant locators which is more than enough.

This method assumes that each emulated IoT device runs on a separate machine with a unique MAC address, which is reasonable as this emulation is designed to be run so that each emulated device runs on a different machine in the same IPv6 network, where each MAC address will be unique.

7.8 Concurrency

7.8.1 Choice of Concurrency Module

The Python concurrency modules which operate in a single process like threading and asyncio do not achieve true parallelism as Python's Global Interpreter Lock prevents threads in the same process from executing simultaneously. In contrast, the multiprocessing module achieves true parallelism by running several processes and this can provide performance benefits, however, these are

most obvious when the program is CPU-bound, and the emulated IoT process is IO-bound as it performs a lot of network IO and no heavy computation. Additionally the use of multiple processes requires complex IPC because processes do not share the same data. Therefore the multiprocessing module was not used in this emulation.[16]

The threading module allows the creation of several threads that execute in an interleaved fashion and like the multiprocessing module, it uses preemptive multitasking so the user does not have much control over when different executions start and stop. Since the different threads operate on shared data, complex concurrency primitives would be required to avoid concurrency bugs like data races. This would lead to fragile code so the threading module was not used.[16]

The asyncio module enables concurrency using an event loop which manages a set of tasks which engage in cooperative multitasking. When a task is waiting for the completion of an event, it can "`await`", and hand control back to the event loop which then chooses another task to execute. This allows the creation of a responsive event-driven model in which tasks await network data and timeouts.[16]

Another advantage of asyncio is that a task is only preempted when the user explicitly "`await`"s, which makes it easier to build robust code as a task will not be preempted at an unexpected point when the program state is inconsistent.

Finally, the usage of asyncio means that no overhead will be incurred in the creation of additional threads or processes which is ideal for resource constrainted IoT devices.

Due to all of the above advantages, the asyncio library was chosen as the concurrency module used to implement the emulated IoTs.

7.8.2 Asyncio Constructs Used

The emuluated IoT simply runs as a series of tasks that engage in cooperative multitasking, and to stop the emulated IoT each of these tasks is cancelled.

When an emulated IoT communicates with other emulated IoTs over the IPv6 channel, a DatagramEndpoint is used which remains dormant until the socket it was passed receives data, at which point the defined protocol which was passed to the DatagramEndpoint executes a callback function that processes datagrams[12]. This allows a responsive event-driven approach to responding to emulation communication while avoiding the overhead of a busy wait. Since sockets cannot bind to multiple addresses without binding to all available addresses, a DatagramEndpoint must be created for each locator the emulated IoT is associated with, and the sockets which underlie these DatagramEndpoints will

be bound to a single IPv6 multicast address.

To implement the transport layer ports, asyncio queues are used to store data received on that port, and the application can asynchronously wait for data arriving on that port. Once this data is received, it is wrapped in an object which contains information about the link layer and network layer source identifers and locators, along with the source port which may be useful in the application.

Additionally, asyncio Events are used for tasks which must wait for an event to happen. An example of this is that the task which retransmits locator updates must wait for the emulated IoT to have an unacknowledged locator update, which is indicated by the setting of an asyncio Event that can be asynchronously awaited on.

7.9 Control Plane and API

While each emulated IoT device runs on a separate machine, an application on a single machine can make API calls by to any number of emulated devices over the IPv4 control channel.

Over this control channel, the application will send API messages which contain the given operation's unique code, and the emulated IoT will perform the corresponding operation and send back a return value. It is important that this control channel runs over TCP so the API connections are reliable.

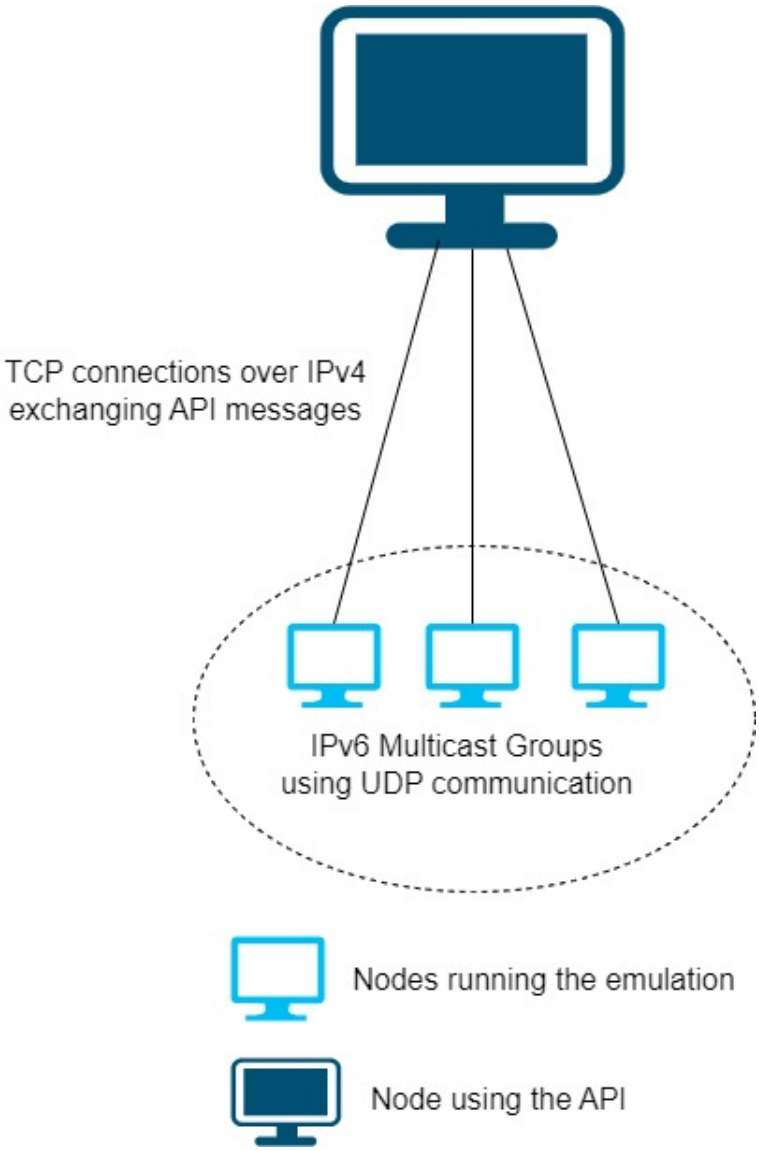

Figure 15: Diagram showing how the emulation uses IPv4 API communication and IPv6 emulation communication

The PyDoc documentation for the API methods is available in the "Emulation API PyDoc" appendix.

Care was taken to follow the end-to-end principle during the development of this API. It says that features can only be properly implemented with the context of the level they operate at, so the best option when designing a supporting API

is to provide a set of primitives which the upper layers can use to implement these features, rather than attempt to provide the feature itself. This is shown by how the example applications utilise this API: they make use of the very basic features this API provides, and combine them to create application specific functionality.

The emulated IoTs use asyncio servers to serve the API calls in a responsive and event-driven manner. The application uses asyncio streams to make the API calls for the same reason. A side effect of this is that if the application must make multiple API calls simultaneously, each of the simultaneous API calls must be made with a different pair of API streams so that no two of the applications calls are waiting on data from the same stream, which would raise an error.

7.10 Management

One of the primary objectives of this project is to provide a management interface for this emulation. This is because just one emulated device is allowed per machine, and logging in to each machine separately to set each emulated device up would be a painstaking and arduous process, and by supplying a usable management interface, the need to perform this laborious task is removed.

Firstly, a script called `EmulationServer.py` is provided. It simply runs the emulation management server on the port supplied as a command line argument, and it handles 3 types of requests, all of which can be made using the API:

- START_IOT_DEVICE: Upon receiving this request, the server starts an emulated IoT device with the supplied network protocol (ILNP or IPv6) and the supplied number of invariant locators (if this parameter is -1, the emulated IoT device does not take part in routing).

  If an emulated IoT device is already running, the request is rejected.

- HANDSHAKE: This request from the API attempts to initiate a handshake with the management server, and is used to check if the server is available to start an emulated IoT. The handshake is rejected if an emulated IoT device is already running. The API call is configured with a timeout so if the server is not running, this is detected.

- CANCEL: Used to cancel the task running the emulated IoT device.

```
djv1@pc8-037-l:~/Documents/CS4099/ilnp_iot-djv1/src $ ./EmulationServer.py 34355
```

Figure 16: Usage of the `EmulationServer.py` script

This script could be run as a service on Linux machines using systemd. This is done by taking the submitted unit file describing the service unit which runs `EmulationServer.py` called `emulated_iot.service` and running the following command to instruct the system to run the service at boot time[28].

```
sudo systemctl enable emulated_iot.service
```

By running the management server as a service which is started upon booting, it will always service API requests to start and run an emulated IoT device with no further intervention required ever agaib. Therefore this is the suggested method for running this emulation.

Unfortunately, this is understandably not permitted in the school lab environment as the amount of persistent state and services which survive rebooting is minimised so that a reboot can fix most problems with a lab computer.

This means that for the purposes of developing in the school labs, the management servers were run in TMUX sessions. Scripts to automate this and restart sessions if they had been ended by a reboot of the machine were developed and instructions on running them are detailed in the User Manual, however the recommended method of running it as a systemd service should be used if possible.

7.11 Configuration

Each emulated IoT device obtains its configuration parameters from a YAML file called `config.yml`. YAML was chosen as it is easy for the user to read and write. The parameters which the emulated IoTs accept in this configuration file are detailed in the User Manual.

7.12 Soft-State Handling

To allow the system to gracefully degrade if a single node fails, soft state was used for many data structures including the forwarding table, ILV cache and the cache storing the valid set of the emulated IoT's own locators. This was implemented by creating a set of tasks which looped throughout the lifetime of the emulated IoT, and checked to see if any entries in the data structure it was checking had expired. To improve performance, these loops used asyncio sleeps.

7.13 Simplifying Assumptions and Decisions

While ILNP allows nodes to have multiple node identifiers, all the emulation API functions can be implemented with emulated IoTs which have just one identifier, so this emulation only emulates IoTs with a single identifier.

This emulation is designed for experimental or recreational use on a network that the user trusts, and not for any kind critical industrial applications or handling of any sensitive data. This justifies the assumption that the emulation will operate in a reduced threat environment, and therefore the nonce destination options which are meant to prevent "off-path attacks" specified in RFC6744[18] are not implemented. Their additional function of indicating that ILNP is in use is redundant in this emulation because the user ensures that every emulated IoT uses the same network layer protocol by simply passing the same argument specifying the network layer protocol the emulated IoT is to use.

RFC6741[22] also states that when a datagram is received and the recipient does not have a record of the node with the source identifier being associated with the source locator, the packet should be dropped unless it has the correct nonce destination option for that node. This is not implemented in the emulation either as it operates in a reduced threat environment where correct nonce values are not required to establish trust between nodes.

Finally, no checksums are implemented as the emulation runs over UDP which uses a checksum. This is in contrast to proper ILNP which will run over any link layer, and sometimes uses ILNP ICMP messages which do have a checksum[21].

8 Licensing

This project makes use of code from two open-source libraries: pyYAML[13] and psutil[30]. The code used from these each of these libraries was moved into single files, which contain the licenses and copyright notices of the corresponding library, in accordance with the terms of use set out by their licenses. The psutil library uses a 3-Clause BSD license, and the pyYAML library uses the MIT license.

The MIT license is compatible with the 3-Clause BSD license, so the rest of the project was licensed with the 3-Clause BSD license with the copyright attributed to Daniel Vallance. The 3-Clause BSD license was chosen as it means that other people may use this project, however they must include the license and copyright notice which was applied. Additionally the disclaimer is quite comprehensive and indemnifies the developer from any damages resulting from others using this project.

9 Testing

The emulation was tested against the requirements which were suitable for testing. The tests are summarised in the table below.

Test No.	Reqs Tested	Test Description	Result
T1	F1, NF7, NF10, NF12, NF14	2 connected IoTs are set up and the ILV cache of the first IoT is obtained using the `get_ilv_cache` call to ensure it received the neighbour advertisements from the other IoT.	Pass
T2	F2, NF7, NF9, NF12, NF14	A network of 2 connected IoTs is set up and the first IoT sends a series of packets to the second. The second will receive the packets and observe the destination locators they were sent on to ensure multipath communication was being used.	Pass
T3	F3, NF7, NF10, NF12, NF14	A routing IoT is set up with neighbour advertisements and routing advertisements sending every 100 seconds. A second IoT is set up before this 100 seconds is up. If possible this second IoT with associate with the locator advertised by the first IoT and its ILV cache will be obtained with the corresponding getter to ensure the solicitations worked.	Pass
T4	F4, NF7, NF10, NF12, NF14	A network with two routing IoTs and a non-routing IoT is set up. The non-routing IoT will join the first routing IoT's locator, then leave it. It will then join it again and replace its locator set with the second routing IoT's locator. Throughout this test, the locators of the non-routing IoT will be obtained with the `get_locs` API call, and the ILV caches of the routing IoTs will be obtained with the `get_ILV_cache` API call to ensure the locator update mechanism worked.	Pass
T5	F5, NF7, NF10, NF12, NF14	A network with a routing IoT and a non-routing IoT is set up. The routers available to the non-routing IoT will be obtained with the `get_available_routers` API call to ensure the router advertisements are working	Pass
T6	F6, NF4, NF7, NF9, NF12, NF14	A network with a routing IoT with two invariant locators and two non-routing IoTs is set up. Each non-routing IoT will associate with a different invariant locator of the routing IoT. The first non-routing IoT should attempt to send a packet to the second non-routing IoT via the routing IoT to ensure the forwarding table is being built up correctly.	Pass

T7	F7, NF4, NF7, NF9, NF10, NF12, NF14	A network with a routing IoT with two invariant locators and two non-routing IoTs is set up. Each non-routing IoT will associate with a different invariant locator of the routing IoT. The first non-routing IoT should attempt to send a packet to the second non-routing IoT via the routing IoT. Then all IoTs apart from one of the non-routing IoTs are shut down. After the timeouts have been triggered, the active correspondents, ILV cache, own locators, available routers and the forwarding table of the remaining IoT are obtained with the relevant API calls to ensure the soft state is being purged correctly	Pass
T8	F8, NF7, NF10, NF14	A single IoT is set up and its identifier value and invariant locators are obtained with the corresponding API getter methods. These values are compared to the MAC address of the machine it is running on to ensure these values are being derived from the MAC address.	Pass
T9	F9, NF7, NF12, NF14	A network is set up with a routing IoT with invariant locators and the IPv6 multicast groups that the underlying machine is a member of will be observed to ensure they correspond to the locators the emulated IoT is associated with.	Pass
T10	F10, NF7, NF9, NF12, NF14	A network with two IoTs, each with an open socket on two ports will be set up. The first will send to the second on both sockets, however the second will only read from one of the ports to ensure that the emulated IoTs can perform transport layer multiplexing and demultiplexing.	Pass
T11	F11, NF4, NF7, NF9, NF10, NF12, NF14	A network with 2 routing IoTs, each with one invariant locator, and two non-routing IoTs is set up. Each routing IoT will associate with the locator of the other routing IoT, and each non-routing IoT will associate with the invariant locator of different IoTs. The first non-routing IoT should attempt to send a packet to the second non-routing IoT. The correspondents of each IoT will be obtained with the `get_correspondents` call. The absence of one of the routers will confirm that the link layer is working correctly.	Pass
T12	F12, NF4, NF7, NF9, NF12, NF14	A network with a non-routing IoT and 2 routing IoTs each with one invariant locator is set up. Each IoT will join every locator, and the non-routing IoT will attempt a transport level flood. The fact that each recipient will only receive one datagram will indicate that no flooding loops occured.	Pass
T13	F16, NF7, NF14	The API calls that perform the management handshake, start an emulated IoT device, get the streams required to communicate with the IoT device, create a socket, join locators, leave locators, replace locators, flood data, send data and quit will be called and the return values will be observed.	Pass

T14	F18, F19, F20, NF7, NF14, NF15, NF16	The emulation server should be started on at least ten IoTs. A handshake will be performed on one of the hosts running the emulation server, then an emulated IoT device will be started on it, and another handshake will be performed. This emulated ILNP IoT will then be shut down with an API call and the handshake will be performed again.	Pass
T15	F22, F23, NF7, NF10, NF12 and NF19	An emulated IoT is started and its CPU usage, RSS and virtual memory used will be obtained with the corresponding API calls.	Pass

Test Evidence

T1

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> i2 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,0,0))
>>> r1,w1 = l.run_until_complete(i1.get_iot_device_conn())
>>> r2,w2 = l.run_until_complete(i2.get_iot_device_conn())
>>> i1loc=l.run_until_complete(i1.get_locs(r1,w1))[0]
>>> i1loc
8939866837170257920
>>> l.run_until_complete(i2.join(r2,w2,i1loc))
True
>>> l.run_until_complete(i2.get_ilv_cache(r2,w2))
{8939866837170257920: {8939866837170257920: <RemoteLocEntry.RemoteLocEntry object at 0x7f49327cef10>}}
>>> l.run_until_complete(i1.get_id(r1,w1))
8939866837170257920
>>>
```

Figure 17: Python session showing that T1 passed

Figure 17 shows that the I-LV cache of i2 which stores the Identifier-Locator mappings of other IoTs contains the correct mapping for i1. This is because once i2 joins the invariant locator of i1, neighbour advertisements from i1 can reach i2, which can use the information in this to build up its I-LV cache, so F1 has been implemented correctly. For this behaviour to be exhibited all the requirements that T1 tests must have been met by the emulation, therefore this test passes.

T2

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,2,0))
>>> i2 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,0,0))
>>> r1,w1 = l.run_until_complete(i1.get_iot_device_conn())
>>> r2,w2 = l.run_until_complete(i2.get_iot_device_conn())
>>> i1locs=l.run_until_complete(i1.get_locs(r1,w1))
>>> i1locs
[8939866837170257920, 8939866837170257921]
>>> l.run_until_complete(i2.join(r2,w2,*i1locs))
True
>>> sock1 = l.run_until_complete(i1.create_socket(3000,r1,w1))
>>> sock2 = l.run_until_complete(i2.create_socket(5000,r2,w2))
>>> id2 = l.run_until_complete(i2.get_id(r2,w2))
>>> id2
8939866837259386880
>>> l.run_until_complete(sock1.send(id2,5000,b'hello',r1,w1))
True
>>> recv_obj = l.run_until_complete(sock2.recv(r2,w2))
>>> recv_obj.prev_hop_loc
8939866837170257920
>>> l.run_until_complete(sock1.send(id2,5000,b'hello',r1,w1))
True
>>> recv_obj = l.run_until_complete(sock2.recv(r2,w2))
>>> recv_obj.prev_hop_loc
8939866837170257921
>>> l.run_until_complete(sock1.send(id2,5000,b'hello',r1,w1))
True
>>> recv_obj = l.run_until_complete(sock2.recv(r2,w2))
>>> recv_obj.prev_hop_loc
8939866837170257920
>>>
```

Figure 18: Python session showing that T2 passed

Once i2's socket calls its receive method to obtain the objects that store the data that i1's socket sent, it observes the `prev_hop_loc` attribute which contains the locator that the receiving IoT was reached on. Figure 18 shows that this locator alternates between the two locators that i2 joined, and this is because the multipath behaviour of the emulation was implemented correctly. For this behaviour to be exhibited all the requirements that T2 tests must have been met by the emulation, therefore this test passes.

T3

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> i2 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,0,0))
>>> r1,w1 = l.run_until_complete(i1.get_iot_device_conn())
>>> r2,w2 = l.run_until_complete(i2.get_iot_device_conn())
>>> i1loc=l.run_until_complete(i1.get_locs(r1,w1))[0]
>>> l.run_until_complete(i2.join(r2,w2,i1loc))
True
>>> l.run_until_complete(i2.get_available_routers(r2,w2))
{8939866837170257920: <AvailableRouterEntry.AvailableRouterEntry object at 0x7f5c1165be80>}
>>> l.run_until_complete(i2.get_ilv_cache(r2,w2))
{8939866837170257920: {8939866837170257920: <RemoteLocEntry.RemoteLocEntry object at 0x7f5c1163eb20>}}
>>> l.run_until_complete(i1.get_id(r1,w1))
8939866837170257920
>>>
```

Figure 19: Python session showing that T3 passed

Figure 19 shows that i2 has i1 recorded as an available router, and has a valid entry for i1 in its I-LV cache. This is in spite of neighbour advertisements and router advertisements being sent just every 100 seconds, and i2 joining i1's locator after the first batch of advertisements has been sent, and its I-LV cache and available routers cache being obtained well before this 100 seconds is up. This must mean that i2 sent neighbour and router solicitations which i1 responded to. For this behaviour to be exhibited all the requirements that T3 tests must have been met by the emulation, therefore this test passes.

T4

```
djv1@pc8-047-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> r1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> r2 = l.run_until_complete(get_iot_device("pc8-053-l",34355,21878,1,0))
>>> nr = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,-1,0))
>>> r1r,r1w = l.run_until_complete(r1.get_iot_device_conn())
>>> r2r,r2w = l.run_until_complete(r2.get_iot_device_conn())
>>> nrr,nrw = l.run_until_complete(nr.get_iot_device_conn())
>>> r1loc = l.run_until_complete(r1.get_locs(r1r,r1w))[0]
>>> r2loc = l.run_until_complete(r2.get_locs(r2r,r2w))[0]
>>> l.run_until_complete(r1.join(r1r,r1w,r2loc))
True
>>> l.run_until_complete(nr.join(nrr,nrw,r1loc))
True
>>> nrid = l.run_until_complete(nr.get_id(nrr,nrw))
>>> r1sock = l.run_until_complete(r1.create_socket(1000,r1r,r1w))
>>> r2sock = l.run_until_complete(r2.create_socket(1000,r2r,r2w))
>>> nrsock = l.run_until_complete(nr.create_socket(1000,nrr,nrw))
>>> l.run_until_complete(r1sock.send(nrid,1000,b'hello',r1r,r1w))
True
>>> l.run_until_complete(r2sock.send(nrid,1000,b'hello',r2r,r2w))
True
>>> l.run_until_complete(nr.leave(nrr,nrw,r1loc))
True
>>> l.run_until_complete(nr.get_locs(nrr,nrw)) == []
True
>>> nrid not in l.run_until_complete(r1.get_ilv_cache(r1r,r1w))
True
>>> nrid not in l.run_until_complete(r2.get_ilv_cache(r2r,r2w))
True
>>> l.run_until_complete(nr.join(nrr,nrw,r1loc))
True
>>> r1loc in l.run_until_complete(nr.get_locs(nrr,nrw))
True
>>> r1_ilvc = l.run_until_complete(r1.get_ilv_cache(r1r,r1w))
>>> r2_ilvc = l.run_until_complete(r2.get_ilv_cache(r2r,r2w))
>>> nrid in r1_ilvc
True
>>> nrid in r2_ilvc
True
>>> r1loc in r1_ilvc[nrid].keys()
True
>>> r1loc in r2_ilvc[nrid].keys()
True
>>> l.run_until_complete(nr.leave(nrr,nrw,r1loc))
```

Figure 20: First part of Python session showing that T4 passed

```
>>> l.run_until_complete(nr.leave(nrr,nrw,r1loc))
True
>>> l.run_until_complete(nr.get_locs(nrr,nrw)) == []
True
>>> nrid not in l.run_until_complete(r1.get_ilv_cache(r1r,r1w))
True
>>> nrid not in l.run_until_complete(r2.get_ilv_cache(r2r,r2w))
True
>>> l.run_until_complete(nr.join(nrr,nrw,r1loc))
True
>>> r1loc in l.run_until_complete(nr.get_locs(nrr,nrw))
True
>>> r1_ilvc = l.run_until_complete(r1.get_ilv_cache(r1r,r1w))
>>> r2_ilvc = l.run_until_complete(r2.get_ilv_cache(r2r,r2w))
>>> nrid in r1_ilvc
True
>>> nrid in r2_ilvc
True
>>> r1loc in r1_ilvc[nrid].keys()
True
>>> r1loc in r2_ilvc[nrid].keys()
True
>>> l.run_until_complete(nr.replace(nrr,nrw,r2loc))
True
>>> r2loc in l.run_until_complete(nr.get_locs(nrr,nrw))
True
>>> r1loc not in l.run_until_complete(nr.get_locs(nrr,nrw))
True
>>> r1_ilvc = l.run_until_complete(r1.get_ilv_cache(r1r,r1w))
>>> r2_ilvc = l.run_until_complete(r2.get_ilv_cache(r2r,r2w))
>>> nrid in r1_ilvc
True
>>> nrid in r2_ilvc
True
>>> r2loc in r1_ilvc[nrid].keys()
True
>>> r2loc in r2_ilvc[nrid].keys()
True
>>> r1loc not in r1_ilvc[nrid].keys()
True
>>> r1loc not in r2_ilvc[nrid].keys()
True
>>>
```

Figure 21: Second part of Python session showing that T4 passed

Figure 20 shows the non-routing IoT joining a locator, then performing unicast sending with the routers so they are in each others' correspondents caches. The fact this send works shows the non-routing IoT joining the locator worked as expected (which could only be possible if the join API call works as expected). Then the non-routing IoT leaves this locator, and the locator does not appear in the non-routing IoT's locator set, or the ILV cache of the routing IoTs which proves the locator udpate mechanism works as expected. The non-routing IoT joins the locator again, and the locator then appears in the non-routing IoT's locator set, and its entry in the ILV caches of the routing IoTs.

Then Figure 21 shows that upon the non-routing IoT leaving again, the state is updated correctly as before. Then another join happens and the state is updated as expected again. Finally a replace call is made, and only the new locator is in the non-routing IoT's locator set, and its entry in the ILV caches of the routing IoTs.

For this behaviour to be exhibited all the requirements that T4 tests must have been met by the emulation, therefore this test passes.

T5

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> i2 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,-1,0))
>>> r1,w1 = l.run_until_complete(i1.get_iot_device_conn())
>>> r2,w2 = l.run_until_complete(i2.get_iot_device_conn())
>>> i1id=l.run_until_complete(i1.get_id(r1,w1))
>>> i1locs=l.run_until_complete(i1.get_locs(r1,w1))
>>> i2id=l.run_until_complete(i2.get_id(r2,w2))
>>> i2available_routers=l.run_until_complete(i2.get_available_routers(r2,w2))
>>> i1available_routers=l.run_until_complete(i1.get_available_routers(r1,w1))
>>> len(i2available_routers)
1
>>> i2available_routers.keys()
dict_keys([8939866837170257920])
>>> i1id
8939866837170257920
>>> i2available_routers[i1id].locs
[8939866837170257920]
>>> i1locs
[8939866837170257920]
>>> len(i1available_routers)
0
```

Figure 22: Python session showing that T5 passed

Despite i2 being a member of no locators, Figure 22 shows it still has an entry for i1 in its available routers cache, with all its invariant locators included. This is because i2 interpreted a router advertisement from i1 on the broadcast locator. For this behaviour to be exhibited all the requirements that T5 tests must have been met by the emulation, therefore this test passes.

T6

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> router = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,2,0))
>>> i1 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,-1,0))
>>> i2 = l.run_until_complete(get_iot_device("pc8-053-l",34355,21878,-1,0))
>>> rr,rw = l.run_until_complete(router.get_iot_device_conn())
>>> i1r,i1w = l.run_until_complete(i1.get_iot_device_conn())
>>> i2r,i2w = l.run_until_complete(i2.get_iot_device_conn())
>>> rlocs = l.run_until_complete(router.get_locs(rr,rw))
>>> l.run_until_complete(i1.join(i1r,i1w,rlocs[0]))
True
>>> l.run_until_complete(i2.join(i2r,i2w,rlocs[1]))
True
>>> sock1 = l.run_until_complete(i1.create_socket(1000,i1r,i1w))
>>> sock2 = l.run_until_complete(i2.create_socket(2000,i2r,i2w))
>>> i2id = l.run_until_complete(i2.get_id(i2r,i2w))
>>> l.run_until_complete(sock1.send(i2id,2000,b'hello',i1r,i1w))
True
>>> l.run_until_complete(sock2.recv(i2r,i2w)).payload
b'hello'
>>>
```

Figure 23: Python session showing that T6 passed

Despite i1 and i2 not sharing any locators, Figure 23 shows that the data sent from i1 reaches i2. This is because i1 knows the locators from which i2 is accessible as per F1, and i1 has also built up a forwarding table from the route that i2's neighbour advertisements took to reach it, and so F6 is satisfied. For this behaviour to be exhibited all the requirements that T6 tests must have been met by the emulation, therefore this test passes.

T7

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> router = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,2,0))
>>> i1 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,-1,0))
>>> i2 = l.run_until_complete(get_iot_device("pc8-053-l",34355,21878,-1,0))
>>> rr,rw = l.run_until_complete(router.get_iot_device_conn())
>>> i1r,i1w = l.run_until_complete(i1.get_iot_device_conn())
>>> i2r,i2w = l.run_until_complete(i2.get_iot_device_conn())
>>> rlocs = l.run_until_complete(router.get_locs(rr,rw))
>>> l.run_until_complete(i1.join(i1r,i1w,rlocs[0]))
True
>>> l.run_until_complete(i2.join(i2r,i2w,rlocs[1]))
True
>>> sock1 = l.run_until_complete(i1.create_socket(1000,i1r,i1w))
>>> sock2 = l.run_until_complete(i2.create_socket(2000,i2r,i2w))
>>> i2id = l.run_until_complete(i2.get_id(i2r,i2w))
>>> l.run_until_complete(sock1.send(i2id,2000,b'hello',i1r,i1w))
True
>>> l.run_until_complete(sock2.recv(i2r,i2w)).payload
b'hello'
>>> l.run_until_complete(router.quit(rr,rw))
True
>>> l.run_until_complete(i2.quit(i2r,i2w))
True
>>> ilv_cache = l.run_until_complete(i1.get_ilv_cache(i1r,i1w))
>>> own_locs = l.run_until_complete(i1.get_locs(i1r,i1w))
>>> available_routers = l.run_until_complete(i1.get_available_routers(i1r,i1w))
>>> forwarding_table = l.run_until_complete(i1.get_forwarding_table(i1r,i1w))
>>> correspondents = l.run_until_complete(i1.get_correspondents(i1r,i1w))
>>> ilv_cache
{}
>>> own_locs
[]
>>> available_routers
{}
>>> forwarding_table
{}
>>> correspondents
[]
>>>
```

Figure 24: Python session showing that T7 passed

Despite i1 and i2 sharing locators, sending data to each other, and all the other conditions for each IoT to have entries involving the other IoT in all the soft state caches that this test deals with, the bottom of Figure 24 shows that these caches are empty. This is because these caches were obtained after waiting for the relevant timeouts to expire therefore soft state is purged correctly.

For this behaviour to be exhibited all the requirements that T7 tests must have

been met by the emulation, therefore this test passes.

T8

```
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,10,0))
>>> r1,w1 = l.run_until_complete(i1.get_iot_device_conn())
>>> i1locs=l.run_until_complete(i1.get_locs(r1,w1))
>>> i1id=l.run_until_complete(i1.get_id(r1,w1))
>>> i1locs
[8939866837170257920, 8939866837170257921, 8939866837170257922, 8939866837170257923,
 8939866837170257924, 8939866837170257925, 8939866837170257926, 8939866837170257927,
 8939866837170257928, 8939866837170257929]
>>> i1id
8939866837170257920
>>> quit()
djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1/src $ ifconfig
eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 138.251.29.217  netmask 255.255.254.0  broadcast 138.251.29.255
        inet6 fe80::7e10:c9ff:fe85:e0d4  prefixlen 64  scopeid 0x20<link>
        ether 7c:10:c9:85:e0:d4  txqueuelen 1000  (Ethernet)
        RX packets 11118754  bytes 10488372751 (9.7 GiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7844092  bytes 5678681723 (5.2 GiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16  memory 0xa2200000-a2220000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 63338  bytes 159381065 (151.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 63338  bytes 159381065 (151.9 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

nebula1: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1300
        inet 10.0.8.55  netmask 255.255.0.0  destination 10.0.8.55
        inet6 fe80::3231:f7c5:84b:f7d  prefixlen 64  scopeid 0x20<link>
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNS
PEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3713  bytes 731636 (714.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1/src $ mac=7c10c985e0d40000
djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1/src $ echo $(( 0x$mac ))
8939866837170257920
```

Figure 25: Python session showing that T8 passed

Figure 25 shows that the identifier, and 20 locators for i1 all use the value 8939866837170257920 as a root, and that this is equal to the MAC address of the underlying machine which hosts i1. Therefore identifiers and locators are

formed using the MAC address of the underlying machine.

For this behaviour to be exhibited all the requirements that T8 tests must have been met by the emulation, therefore this test passes.

T9

```
djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1 $ ip maddr | grep 7c10:c985:e0d4
djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1 $ cd src
djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> router = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,2,0))
>>> r,w = l.run_until_complete(router.get_iot_device_conn())
>>> locs = l.run_until_complete(router.get_locs(r,w))
>>> locs
[8939866837170257920, 8939866837170257921]
>>> quit()
djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1/src $ ip maddr | grep 7c10:c985:e0d4
        inet6 ff02::7c10:c985:e0d4:1
        inet6 ff02::7c10:c985:e0d4:0
djv1@pc8-055-l:~/Documents/CS4099/ilnp_iot-djv1/src $
```

Figure 26: Python session showing that T9 passed

Figure 26 shows that upon joining 2 locators, the underlying machine that hosts the emulated IoT joins 2 IPv6 multicast groups whose least significant 64 bits are the locators values that were joined.

For this behaviour to be exhibited all the requirements that T9 tests must have been met by the emulation, therefore this test passes.

T10

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> i2 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,-1,0))
>>> r1,w1 = l.run_until_complete(i1.get_iot_device_conn())
>>> r2,w2 = l.run_until_complete(i2.get_iot_device_conn())
>>> r1loc = l.run_until_complete(i1.get_locs(r1,w1))[0]
>>> l.run_until_complete(i2.join(r2,w2,r1loc))
True
>>> i1sock = l.run_until_complete(i1.create_socket(1000,r1,w1))
>>> i2sock1 = l.run_until_complete(i2.create_socket(1000,r2,w2))
>>> i2sock2 = l.run_until_complete(i2.create_socket(2000,r2,w2))
>>> i2id = l.run_until_complete(i2.get_id(r2,w2))
>>> l.run_until_complete(i1sock.send(i2id,1000,b'hello on port 1000',r1,w1))
True
>>> l.run_until_complete(i1sock.send(i2id,2000,b'hello on port 2000',r1,w1))
True
>>> l.run_until_complete(i1sock.send(i2id,1000,b'hello on port 1000',r1,w1))
True
>>> l.run_until_complete(i2sock2.recv(r2,w2)).payload
b'hello on port 2000'
>>>
```

Figure 27: Python session showing that T10 passed

Figure 27 shows that despite i1 sending data to port 1000 before and after it send data to port 2000, i2 can still obtain the data sent to port 2000 first due to transport level multiplexing and demultiplexing.

For this behaviour to be exhibited all the requirements that T10 tests must have been met by the emulation, therefore this test passes.

T11

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> r1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> r2 = l.run_until_complete(get_iot_device("pc8-052-l",34355,21878,1,0))
>>> nr1 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,-1,0))
>>> nr2 = l.run_until_complete(get_iot_device("pc8-053-l",34355,21878,-1,0))
>>> r1r,r1w = l.run_until_complete(r1.get_iot_device_conn())
>>> r2r,r2w = l.run_until_complete(r2.get_iot_device_conn())
>>> nr1r,nr1w = l.run_until_complete(nr1.get_iot_device_conn())
>>> nr2r,nr2w = l.run_until_complete(nr2.get_iot_device_conn())
>>> r1loc = l.run_until_complete(r1.get_locs(r1r,r1w))[0]
>>> r2loc = l.run_until_complete(r2.get_locs(r2r,r2w))[0]
>>> l.run_until_complete(r1.join(r1r,r1w,r2loc))
True
>>> l.run_until_complete(r2.join(r2r,r2w,r1loc))
True
>>> l.run_until_complete(nr1.join(nr1r,nr1w,r1loc))
True
>>> l.run_until_complete(nr2.join(nr2r,nr2w,r2loc))
True
>>> sock1 = l.run_until_complete(nr1.create_socket(1000,nr1r,nr1w))
>>> sock2 = l.run_until_complete(nr2.create_socket(2000,nr2r,nr2w))
>>> nr2id = l.run_until_complete(nr2.get_id(nr2r,nr2w))
>>> l.run_until_complete(sock1.send(nr2id,2000,b'hello',nr1r,nr1w))
True
>>> l.run_until_complete(sock2.recv(nr2r,nr2w)).payload
b'hello'
>>> r1correspondents = l.run_until_complete(r1.get_correspondents(r1r,r1w))
>>> r2correspondents = l.run_until_complete(r2.get_correspondents(r2r,r2w))
>>> nr1correspondents = l.run_until_complete(nr1.get_correspondents(nr1r,nr1w))
>>> nr2correspondents = l.run_until_complete(nr2.get_correspondents(nr2r,nr2w))
>>> r1correspondents
[]
>>> r2correspondents
[8939866837259386880, 8939866837112258560]
>>> nr1correspondents
[8939866837096267776, 8939866837112258560]
>>> nr2correspondents
[8939866837096267776, 8939866837259386880]
```

Figure 28: Python session showing that T11 passed

Figure 28 shows that after nr1 sends to nr2, and therefore unicast ILNP communication takes place, nr1, nr2 and r2 have each other as correspondents, while r1 is not a correspondent of any other IoT, and it has no correspondents. This must mean that the data got sent through r2, and despite r1 being a member of all the locators that r2 is, when the application data got sent through this locator, the existence of a functioning link layer meant that r1 dropped the packet containing the application layer data.

For this behaviour to be exhibited all the requirements that T11 tests must have been met by the emulation, therefore this test passes.

T12

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> routing1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> routing2 = l.run_until_complete(get_iot_device("pc8-054-l",34355,21878,1,0))
>>> nonrouting = l.run_until_complete(get_iot_device("pc8-053-l",34355,21878,-1,0))
>>> r1,w1 = l.run_until_complete(routing1.get_iot_device_conn())
>>> r2,w2 = l.run_until_complete(routing2.get_iot_device_conn())
>>> r3,w3 = l.run_until_complete(nonrouting.get_iot_device_conn())
>>> r1loc = l.run_until_complete(routing1.get_locs(r1,w1))[0]
>>> r2loc = l.run_until_complete(routing2.get_locs(r2,w2))[0]
>>> l.run_until_complete(routing2.join(r2,w2,r1loc))
True
>>> l.run_until_complete(routing1.join(r1,w1,r2loc))
True
>>> l.run_until_complete(nonrouting.join(r3,w3,r1loc,r2loc))
True
>>> sock1 = l.run_until_complete(routing1.create_socket(1000,r1,w1))
>>> sock2 = l.run_until_complete(routing2.create_socket(1000,r2,w2))
>>> sock3 = l.run_until_complete(nonrouting.create_socket(1000,r3,w3))
>>> l.run_until_complete(sock3.flood(1000,b'hello',r3,w3))
True
>>> l.run_until_complete(sock1.recv(r1,w1)).payload
b'hello'
>>> l.run_until_complete(sock1.has_data(r1,w1))
False
>>> l.run_until_complete(sock2.recv(r2,w2)).payload
b'hello'
>>> l.run_until_complete(sock2.has_data(r2,w2))
False
>>> l.run_until_complete(sock3.has_data(r3,w3))
False
```

Figure 29: Python session showing that T12 passed

Figure 29 shows that despite the three IoTs being in a triangular topology in which unbounded flooding could occur, just one instance of the flooded data is available to the recipient IoTs' sockets.

For this behaviour to be exhibited all the requirements that T12 tests must have been met by the emulation, therefore this test passes.

T13

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> l.run_until_complete(handshake("pc8-055-l", 34355))
True
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> r,w = l.run_until_complete(i1.get_iot_device_conn())
>>> sock = l.run_until_complete(i1.create_socket(1000,r,w))
>>> l.run_until_complete(i1.join(r,w,1))
False
>>> l.run_until_complete(i1.leave(r,w,1))
True
>>> l.run_until_complete(i1.replace(r,w,1))
False
>>> l.run_until_complete(sock.flood(1000, b'hello',r,w))
True
>>> l.run_until_complete(sock.send(1000, 1234, b'hello',r,w))
False
>>> l.run_until_complete(i1.quit(r,w))
True
>>>
```

Figure 30: Python session showing that T13 passed

Figure 30 shows that:

- The handshake API call returns True if the host is available to host an emulated IoT.
- The `get_iot_device` API call returns an iot device object if successful
- The `get_iot_device_conn` API call returns the streams necessary to communicate with the control plane of the emulated iot device if successful
- The `create_socket` method returns a socket object if successful
- Passing a nonexistent locator to the join API call returns false as a nonexistent locator cannot be joined
- Passing a nonexistent locator to the leave API call returns true as the emulated IoT was not associated with this locator in the first place
- Passing a nonexistent locator to the replace API call return false as a nonexistent locator cannot be joined
- Calls to the flood API call return True when successful. The condition for success is that the data is sent on all locators, and this can be true even when the emulated IoT is not associated with any locators

- Sending to a nonexistent IoT cannot be done, and attempting to do so with the send API call returns false.
- Ending the emulated IoT with a call to the quit API call returns True when successful

For this behaviour to be exhibited all the requirements that T13 tests must have been met by the emulation, therefore this test passes.

T14

```
djv1@pc8-037-l:~/Documents/CS4099/ilnp_iot-djv1 $ ./refresh_tmux_servers.sh hostnames 34355 ~/Documents/CS4099/ilnp_iot-djv1/
pc8-037-l
pc8-038-l
pc8-040-l
no server running on /tmp/tmux-21878/default
pc8-041-l
ssh: connect to host pc8-041-l.cs.st-andrews.ac.uk port 22: No route to host
pc8-042-l
pc8-043-l
pc8-044-l
no server running on /tmp/tmux-21878/default
pc8-045-l
pc8-046-l
server exited unexpectedly
pc8-047-l
pc8-048-l
pc8-049-l
pc8-050-l
server exited unexpectedly
pc8-051-l
pc8-052-l
pc8-053-l
pc8-054-l
pc8-055-l
```

Figure 31: Running of script to set up emulation management servers in TMUX sessions

```
djv1@pc8-057-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> l.run_until_complete(handshake("pc8-037-l",34355,5))
True
>>> iot = l.run_until_complete(get_iot_device("pc8-037-l",34355,21878,1,0))
>>> l.run_until_complete(handshake("pc8-037-l",34355,5))
False
>>> r,w = l.run_until_complete(iot.get_iot_device_conn())
>>> l.run_until_complete(iot.quit(r,w))
True
>>> l.run_until_complete(handshake("pc8-037-l",34355,5))
True
>>>
```

Figure 32: Python session showing that T14 passed

Figure 31 shows the initialisation of TMUX sessions running the emulation management server on hosts in the school lab machines. Not all of them will be up, however that is ok as a timeout mechanism is employed.

Figure 32 shows that when there is not a device running on it, the handshake with the emulation management server returns true. Then, once a device is successfully initialised on it, it returns false. Once this emulated IoT is shut down however, the handshake returns true to indicate it is available to run a new emulated IoT.

For this behaviour to be exhibited all the requirements that T14 tests must have been met by the emulation, therefore this test passes.

T15

```
djv1@pc8-039-l:~/Documents/CS4099/ilnp_iot-djv1/src $ python
Python 3.9.14 (main, Jan  9 2023, 00:00:00)
[GCC 11.3.1 20220421 (Red Hat 11.3.1-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from emulation_api import *
>>> l = asyncio.new_event_loop()
>>> i1 = l.run_until_complete(get_iot_device("pc8-055-l",34355,21878,1,0))
>>> r1,w1 = l.run_until_complete(i1.get_iot_device_conn())
>>> l.run_until_complete(i1.get_cpu_time(r1,w1))
0.13
>>> l.run_until_complete(i1.get_rss(r1,w1))
21909504
>>> l.run_until_complete(i1.get_virtual_mem(r1,w1))
252497920
>>>
```

Figure 33: Python session showing that T15 passed

Figure 33 proves that the API calls which obtain resource usage data of the emulated IoT function correctly.

For this behaviour to be exhibited all the requirements that T15 tests must have been met by the emulation, therefore this test passes.
All of the tests have passed.

10 Fulfilment of Requirements

The "Testing" section above demonstrated how most of the requirements have been met, however there are a few requirements which were not suitable for testing, so this section contains explanations of how these requirements were met.

Requirement	How it was met
F13	The emulation code uses asynchronous programming as the asyncio library is used extensively
F14	The API calls are made using IPv4 as the API calls and responses are sent to and from IPv4 addresses
F15	The API code uses asynchronous programming as the asyncio library is used extensively
F17	The control interface is implemented in API calls as various control mechanisms such as adding to, removing from and replacing locator sets are implemented using API calls.
F21	The emulation is packaged with a requirements.txt file which can be found in the src/ directory
NF1	The ILNP emulation clearly uses an addressing split as identifiers and locators are handled separately
NF2	The emulated IoTs self-configure by sending neighbour and router solicitations when they start up, and interpreting the responses they get. Intuitively, this requirement is met as the IoTs achieve connectivity without the user having to directly input any connectivity information (except when the user wants to make specific changes to an emulated IoT's locator set).
NF3	There is no additional infrastructure in this emulated IoT network apart from the emulated IoTs therefore it is an ad-hoc network.
NF5	The emulation demonstrates its robustness and its ability to exchange high volumes of data in mobility and multipath scenarios in the experimental scenario applications
NF6	The emulation demonstrates its scalability and its ability to run applications involving 10 or more IoTs in the experimental scenario applications
NF8	The emulated IoTs take their configurable parameters from the config.yml file
NF11	This requirement was met through the usage of the asynchronous programming
NF13	The experimental scenario applications demonstrate the management interface's ability to set up topologies defined in YAML files
NF17	The emulation is packaged with open source licensing as discussed in the "Licensing" section
NF18	The User Manual in contained in an appendix, and in the submitted code folder. The README is contained in the submitted code folder.

From the "Testing" section above, and the justifications in the table above, it is clear that every requirement has been met.

11 Experimental Scenario Applications

The aim of this project is to evaluate the relative suitability of ILNP-enabled IoT devices for ubiquitous communication in comparison to IPv6-enabled IoT devices. To this end, a set of scenario applications which incorporate increasingly more mobility and multipath communication were designed and implemented using the emulation's API. The ILNP emulation's performance will be compared to that of a similar emulation which emulates IPv6 devices, and from these results the relative suitability of each protocol for ubiquitous communication can be determined.

11.1 IPv6 emulation

Adapting the ILNP emulation to emulate IPv6-enabled IoT devices instead was made simpler by the fact ILNPv6 is designed to be a superset of IPv6. This means ILNP addresses look identical to IPv6 addresses on the wire, and that IPv6 control messages such as router advertisements can be used by ILNP too, therefore no changes to the emulation's packet structures were required.

The ILNP locator value has the same semantics as the IPv6 routing prefix. However, the semantics of the ILNP node identifer differ from that of the IPv6 Interface Identifier, as they uniquely identify nodes and interfaces respectively[21]. Industry-grade implementations of these protocols are required to handle this difference in semantics at the end systems, however this emulation does not as it only emulates nodes which have just one interface.

Note that this simplification also means the ILNP emulation's API can remain unchanged for the adapted IPv6 version, as methods that require the ILNP node identifier can use the IPv6 interface identifier to the same effect.

The only changes that are required to emulate IPv6 as opposed to ILNP are that the IPv6 emulated nodes must use a mechanism equivalent to the hard handoff mechanism instead of soft handoff, and the multipath functionality must be disabled.

This is because in IPv6, each transport level connection is identified by the whole IPv6 address. Therefore, when connectivity is changed and a new address is obtained, the old connection must be destroyed before a new connection with the new address is created. This is equivalent to an ILNP hard handoff, however as an implementation convenience, other nodes will be informed with a neighbour advertisement after the change in connectivity instead of a hard locator update.

While there are IPv6 mechanisms for ameliorating the disruption to the transport level connection during connectivity changes such as Binding Updates and Mobile IPv6, this is not implemented as it is only standard IPv6 that is being

evaluated here.

Additionally, the IPv6 emulation cannot support multipath communication in the same way as the ILNP emulation. This is because the ILNP emulation implements this by using a deficit round robin algorithm to choose the locator of the remote node to use for each packet, and doing this using IPv6 routing prefixes is impossible for a single transport level connection as the routing prefixes are part of the transport connection identifier tuple. Therefore the IPv6 emulation was adapted so that for a single connection, nodes use the same source and destination prefixes until they are no longer valid or reachable, at which point another connection with different routing prefixes is created.

The disallowing of soft handoff and multipath communcation results in a simplified emulation of IPv6 devices which uses exactly the same API.

11.2 Common Experimental Features

Below are some design and implementation features which span all the experimental scenario applications.

11.2.1 Topology Setup

Each experiment requires multiple emulated IoT devices, initially arranged in some predefined topology. This is achieved with the `create_topology` API call.

This API call takes an application configuration file name, and a network layer protocol to emulate. This configuration file will contain fields called `topology_file` and `hosts_file` which refer to a YAML file that defines the topology of the network to be set up, and a file that contains the list of hostnames which should be checked to see if the management server is running respectively.

The API call creates the initial topology by iterating through each named node in the YAML topology file. It attempts to initialise a new emulated IoT for this named node by calling the `get_running_iot` method. This management plane method iterates through the hosts listed in the hostname file and attempts a handshake with the management server running on that host, and if successful attempts to initialise an emulated IoT device on that host, with the number of invariant locators specified for that named node in the topology file (or the node can be initialised as a non-routing node if the topology file dictates this). If none of the supplied hosts have the management server running, the other emulated nodes are shut down and the API call fails.

Once the emulated node has been initialised with the same number of invariant locators as specified in the YAML topology file, its actual locator values are obtained so they can be mapped to the numbers that the topology file uses to

refer to them. Then if later nodes use the number that the topology file uses to refer to that locator in its `joined` element, it will find the actual locator value through the locator mapping that has been set up, and it will join it.

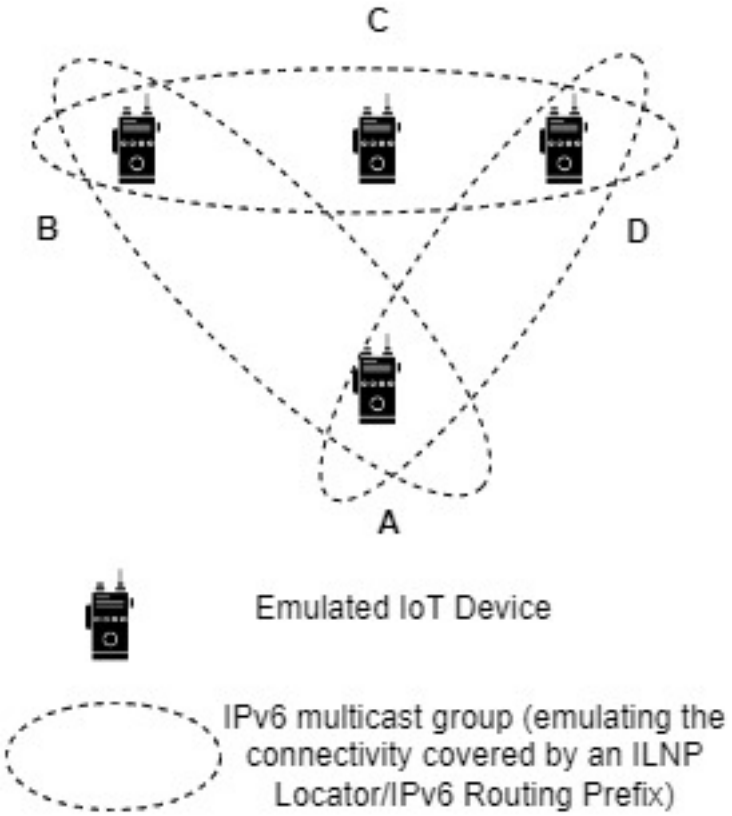

Figure 34: Example topology of emulated IoT devices

An example topology file and API call to set up the topology show in Figure 34 will now be demonstrated.

```
topology:
  A:
    inv: [1,2]
    joined: []
  B:
    inv: [3]
    joined: [1]
  C:
```

```
    joined: [3]
  D:
    inv: []
    joined: [2,3]
```

This example topology YAML file defines A as a routing node with 2 invariant locators and B as a routing node with 1 invariant locator that joins one of A's invariant locators. C is not a routing node as it has no `inv` element, however it does join B's invariant locator, labelled 3. Finally, D is a routing node that has no invariant locators as it has an `inv` element but it is empty, and it joins an invariant locator of A, and an invariant locator of B.

The API call to set up the topology above is as follows:

```
await create_topology(config.yml, NetProtocol.ILNP.value)
```

where config.yml is a YAML file that contains fields referring to the topology file, the hostnames file and parameters that will be used to set up the emulated node, and NetProtocol.ILNP.value is the enum used to refer to ILNP as opposed to IPv6. It is defined in the constants file and has a value of 0.

Once this process has been completed for every node, all the required emulated IoT devices will have been initialised and will be arranged in the specified initial topology, and the experiment may commence.

11.2.2 Simulation of Delay

In order to better emulate the performance of wireless IoT networks over the school labs' ethernet network, some delay should be induced.

Ideally, the `netem` facility would be used to introduce delay between the emulated nodes, however in order to be considerate of others using the school lab machines, an artificial delay was introduced between processes running the emulation.

This was achieved by instead of immediately processing datagrams which arrive at an emulated node, spawning off a task which asynchronously waits 10ms before processing the datagram.

10ms was chosen as the delay value because of the data presented in the paper entitled "Modeling and Comparison of Delay and Energy Cost of IoT Data Transfers"[33]. It presents measurements of the delays incurred by different IoT technologies when sending different amounts of data over different ranges. Over the short and medium ranges, sending 100 bytes or less with all of the evaluated IoT technologies and hardwares resulted in a delay of 10ms or less. Over the

long range, only a single technology resulted in a delay of over 10ms when sending 100 bytes or less as it attempts to reliably send these bytes by repeating the transmission[33]. Since the application packets sent in these scenarios are very small, and each link frame is less than 100 bytes, this value of 10ms models a wide range of IoT technologies and hardwares well.

This delay can be changed in the configuration file.

11.2.3 Emulation Configuration

Each experiment will have its own application-specific parameters which will need to be set, however they will also all supply the emulated IoT devices with configuration parameters in a configuration file. To enforce consistency across the emulated network between experiments, the same emulation configuration is used between experiments. The configuration used is shown below:

```
mcast_prefix: ff02:0000:0000:0000
emulation_port: 21878
mgmt_port: 21878
router_ad_interval: 1
neighbour_ad_interval: 0.1
lu_resend_interval: 0.05
purge_rate_limit: 0.01
own_loc_active_timeout: 5
own_loc_aged_timeout: 5
correspondents_timeout: 5
forwarding_entry_timeout: 0.25
remote_loc_active_timeout: 0.25
remote_loc_aged_timeout: 1.5
remote_loc_expired_timeout: 1.5
available_router_timeout: 5
BROADCAST_LOC: 0xffffffffffffffff
BROADCAST_IDENTIFIER: 0xffffffffffffffff
QUANTUM: 1
FLOW_INFO: 0
SCOPE_ID: 2
last_used_timeout: 1.5
startup_wait: 2
delay: 0.01
max_locs: 140
transport_mss: 1156
```

Every time parameter is expressed in seconds, in both the emulation configurations, and the application configurations.

The neighbour advertisements are set to be flooded very regularly with every node sending one every 0.1 seconds, as these contain the current locator set of a

node which is a volatile piece of state. The router advertisements are broadcast less regularly with every router sending one a second because these only contain the invariant locators it provides access to which does not change during the run of an experiment. Solicitations can be used by a node to demand configuration information as and when they wish, so the regularity of advertisements will not affect how quickly a node can obtain connectivity configuration information.

The locator update resend interval is set to 0.05 seconds to allow for soft handoffs which are fast, while allowing for the fact it may have to go through several nodes each of which will incur a 0.01 second delay
.
The timeouts for active remote locators and forwarding entries are set lower than the timeouts for a node's own active locators and available routers as the neighbour advertisements which inform the ILV cache and forwarding table are sent more regularly than the router advertisements which inform which locators can be used as a node's own locators, so it takes less time to miss the critical number of neighbour advertisements which suggest connectivity is lost, than that for router advertisements.

However, the timeouts for remote locators which are expired or aged are set a bit higher at 1.5s as it is desirable to keep these for a bit longer as a last resort to attempt connectivity.

The correspondents timeout is set to be 5 seconds, as a higher value is desired to ensure that soft handoff is performed with as many correspondent nodes which are likely to communicate with the mobile node in future as possible.

A description of the semantics of these configuration values can be found in the user manual.

11.2.4 Reporting of Emulated IoT Devices' Statistics

To satisfy the secondary objective requiring that the emulation must provide a mechanism for monitoring resource usage, API calls were implemented to obtain the CPU usage, resident set size, and the virtual memory of the process running the emulated IoT device.

Additionally, API calls to obtain the network and application, datagrams and bytes, sent and received, of an emulated IoT device were implemented too. The network data sent is defined as everything that a node transmits and the network data received is everything that a node receives. The application data sent is everything that a node sends with a transport header, and the application data received is everything that a node receives with a transport header. Note that this means that when an application packet is being forwarded, only the sender and receiver register increases in application data sent and received, and any intermediary routers do not.

Each experiment makes use of these API calls and sets up an asynchronous task which regularly requests these values from the emulation and stores them in a data structure which can then be obtained after the experiment's conclusion for analysis.

11.2.5 Communication Between Application Objects and the Emulated IoT Devices

Each application object will have several streams connecting it to its emulated IoT device so it may run several API calls simultaneously, and not encounter the error which is raised when two tasks are awaiting data from the same stream simultaneously.

11.2.6 Experimental Assumptions

The delay calculations are made with the assumption that the clocks of all the lab computers are synchronised.

Additionally, the experiments assume that asynchronous waits are timed perfectly and that send operations happen instantaneously. Under this assumption, sending 6000 datagrams at a rate of 100 datagrams a second should take 60s, however this assumption does not fully hold which results in each experiment taking slightly longer than it should.

11.2.7 Statistical Handling

Each experiment will be run 15 times for each network layer protocol. This number of repeats was chosen as repeating 10 times resulted in more outliers than was desirable, but running 15 times reduced the number of outliers to an acceptable amount.

Each run will be 60 seconds long.

Each experiment will measure a metric related to application layer delay. To present these findings, every delay measurement over every run of each experiment will be placed into a boxplot. Since this is the only measurement that takes every reading from every run, outliers were not shown as this cluttered the graph too much.

Additionally each experiment will measure a metric related to application layer delivery rate. Each run will have its delivery rate measurement calculated, then the final results for each run will be put into a boxplot.

Every object in every experiment will have an asynchronous task that calls the API functions which return resource usage statistics of its emulated IoT device every second. Every experiment will choose the node that performs the most resource-consuming tasks related to ubiquitous communication and analyse its resource consumption data.

The CPU value returned by the corresponding API call is the cumulative sum of the time the CPU has spent in user and system mode for the emulated IoT's process. Therefore each run will have the final CPU time value placed into a boxplot to summarise CPU usage.

The RSS (which is a measure of the physical main memory used by a process) and virtual memory API calls just report the size of physical and virtual memory used by that process at that moment in time. These values show very similar trends so to prevent repeating analysis, only the RSS measurements were analysed, as IoT devices have a limited amount of physical memory so it would be interesting to see if the physical memory used by each protocol makes it unsuitable for IoT devices. Each run has its maximum RSS value placed in the boxplot as knowing the maximum memory used, and therefore the memory required is more useful than knowing the final memory measurement recorded.

Graphs which show how CPU usage and RSS change across a single run for the chosen node for each experiment will be presented too. This may provide some insight on the impact that certain events have on resource usage, however a single run will not be enough to draw any conclusions.

For each run, the total number of network and application bytes and packets sent across the entire network is calculated. This is achieved by every object in every experiment having an asynchronous task which calls the API function that returns overhead statistics, and the results of this can be used to find the total number of network and application bytes and packets each node sent, and the sum of these values for each node will result in the network wide number of network and application bytes and packets sent. Then from this, each run can calculate its overall packet overhead and byte overhead by finding the ratio of the total network measurement to the total application measurement which can be put in a boxplot.

Additionally, byte overhead and packet overhead per second is plotted for a single run by getting the network measurement and application measurement for each second, and finding their ratio.

While observing how these values change in a single run is worth including and may provide insight in how certain events affect a value during a run, the conclusions will be drawn from the data that is derived from all 15 runs.

Finally, note that all the graphs showing a single run of a protocol for an ex-

periment show the same single run.

11.3 Node-Centric Temperature Sensor Network

11.3.1 Description

The first scenario for which an experimental scenario application will be developed using the emulation API is a standard temperature sensor network. It consists of ten emulated IoT sensors, and one emulated IoT sink, and in this case the sensors will push their readings to the single sink at regular intervals. The readings are simply generated by obtaining a random integer between 0 and 99 inclusive.

The initial topology for this application is shown in figure 35 and it also shows that there are multiple paths to the sink for each sensor, therefore there is potential for multipath communication in this application. The emulation of ILNP-enabled IoT devices exploits this as the send primitive provided by the API for ILNP nodes uses a deficit round robin algorithm to distribute the load across the sink's 2 locators.

However, this application does not have any potential for mobility as the nodes are stationary and the reduced duty cycle that is often observed in wireless sensors which invalidates paths and results in connectivity changes similar to those observed in mobile scenarios is ignored[35].

The topology of the emulated IoT nodes in this application is shown in the diagram below:

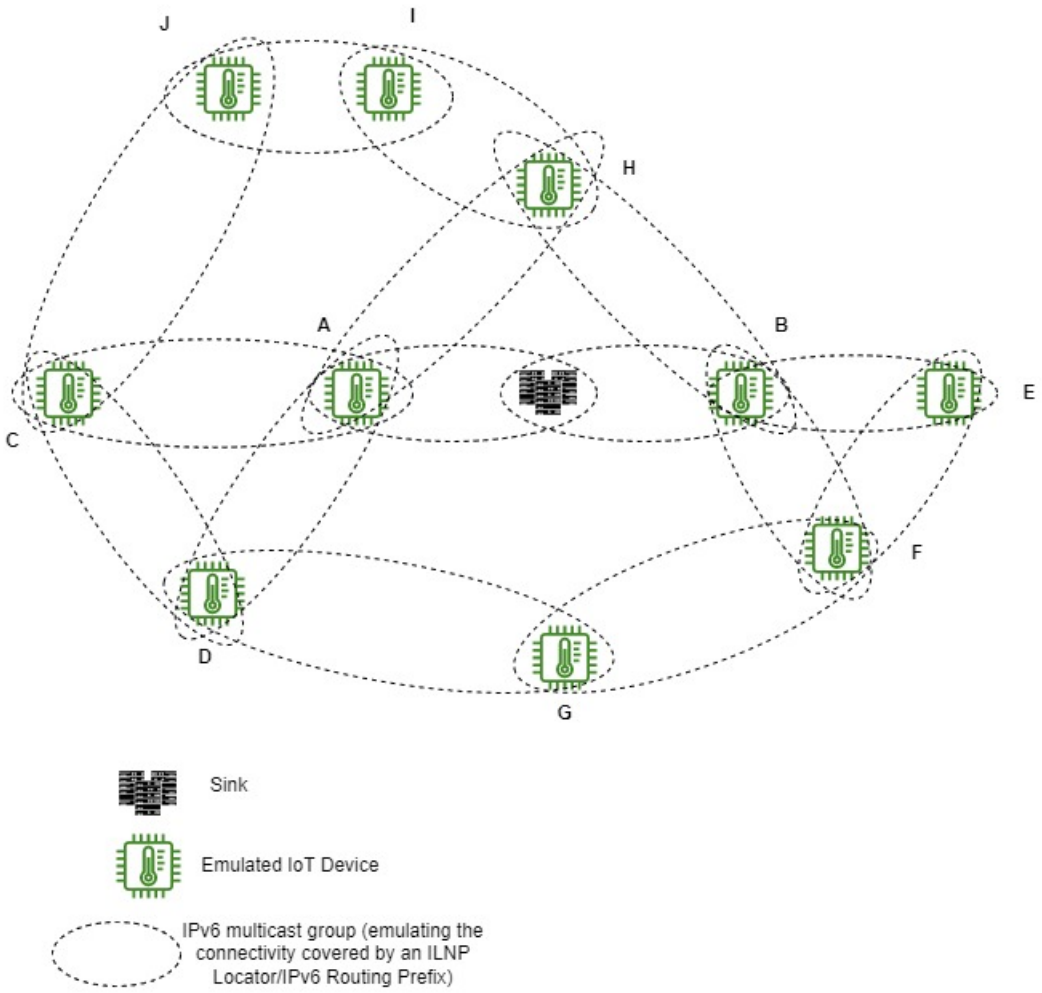

Figure 35: Topology used for the nodecentric and datacentric temperature sensor network applications.

11.3.2 Configuration

The nodecentric temperature sensor network is run with the following application configuration:

```
app_port: 3000
server_port: 34355
emulation_port: 21878
res_interval: 1
sensor_send_interval: 0.01
```

```
handshake_timeout: 2
hosts_file: ../../../hostnames
readings_to_send: 6000
resource_reporting_sensor: f
final_wait: 0.05
topology_file: topology.yml
```

The semantics of the fields most relevant to the operation of the application are as follows:

- `res_interval` is the number of seconds between each object's requests for resource and overhead statistics from their emulated IoT device.
- `sensor_send_interval` is the time period between which successive readings are transmitted by each sensor.
- `readings_to_send` is the number of readings each sensor will send.
- `resource_reporting_sensor` is the arbitrarily chosen sensor whose resource consumption will be graphed and analysed.

11.3.3 Implementation

The identifier of the sink is obtained after its IoT device is set up using the `get_id` API call, and this identifier is passed to each sensor device, which passes this identifier to the send API call when it wishes to publish its readings.

The reading datagrams contain a sequence number so that delivery rate for each sensor can be measured. This is achieved by the sink recording the sequence number of each reading it receives from each sensor, and then the delivery rate for each sensor can be obtained by dividing the number of readings it received from each sensor by the number of readings it sent (as set in the experiment configuration file).

The reading datagrams also contain a timestamp which is set when the sensor sends its reading, so that the one-way delay may be recorded by taking the timestamp away from the time when it is received by the sink.

Below is a packet diagram for the temperature readings.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Temperature
Sequence Number
Timestamp (ms)
Timestamp (ms) continued

Figure 36: Temperature reading packet structure

11.3.4 Method

In this application, the following metrics are recorded:

- CPU usage of the process running a sensor's emulated IoT device.
- The resident set size of the process running a sensor's emulated IoT device.
- One-way delay incurred for each temperature reading, grouped by the source sensor.
- Delivery rate of the temperature readings for each source sensor.
- The packet overhead throughout the emulated network.
- The byte overhead throughout the emulated network.

This is achieved by running the `run_experiment.py` script from the `src/experiments/tsn/` folder which sets up the emulated IoT devices and arranges them in the initial topology as specified earlier, then initialises the Sink and Sensor objects. Its usage is as follows:

```
./run_experiment <app_config_file> <paradigm> <protocol> <no_of_runs>
```

where the protocol is either "ilnp" or "ipv6", and the paradigm can be "nodecentric" or "datacentric", however for this experiment it will be set to "nodecentric".

Once all the Sink and Sensor objects have been set up, the application may begin. Each sensor simply sends a reading to the sink every `sensor_send_interval` seconds, and continues to do so until all `readings_to_send` temperature readings have been sent, then it stops.

Since this application uses a nodecentric paradigm[35], the sink does not need to send anything and it just processes every reading it gets by recording the delay, sequence number and source sensor, which can be used to generate graphs showing delay and delivery rate.

While not part of the application, in order to record the overhead statistics and resource statistics, the sensors and the sinks have tasks which request these values from their emulated IoT device using the API every `res_interval`. These values are processed as stated in the statistical handling section. An arbitrary sensor was chosen as the node that had its CPU and RSS measured, as the sensors send data unlike the sink, and it would be interesting to see how engaging in this communication affects resource consumption.

11.3.5 Results

This section presents the graphs of the data which were created in accordance with the steps outlined in the statistical handling section, along with associated observations and analysis.

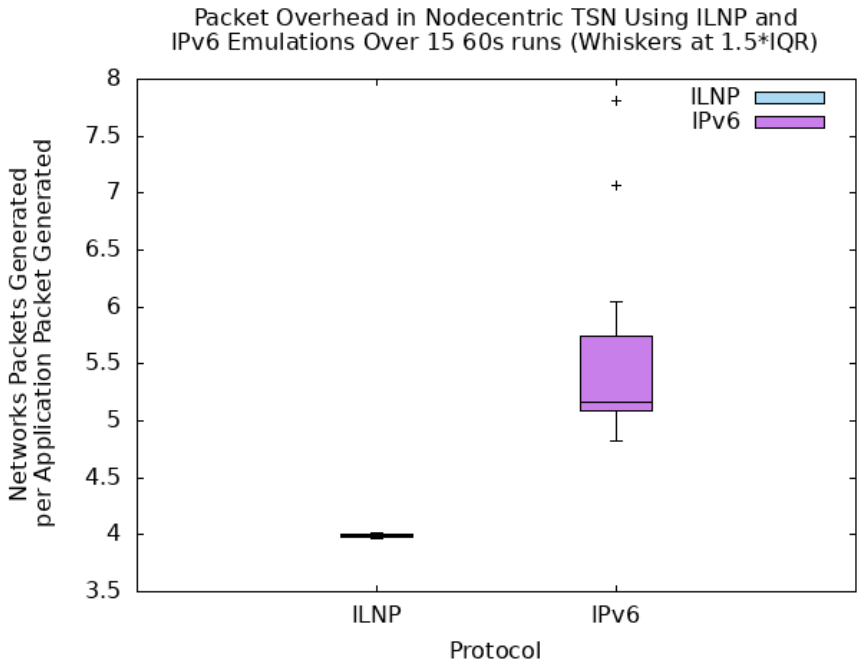

Figure 37: Graph of Packet Overhead in Nodecentric TSN Over 15 Runs

Observations

This graph clearly shows that when using emulated IPv6, the nodecentric TSN application incurs a higher packet overhead than when it uses emulated ILNP. When using emulated ILNP, the nodecentric TSN very consistently generates

4 network packets per application packet over the whole run, whereas when it uses emulated IPv6, the median number of network packets generated per application packet is around 5, however there are outliers in which it almost reaches 8.

Analysis

There is a clear pattern in which using emulated IPv6 incurs more packet overhead for the nodecentric TSN than when it uses emulated ILNP. This is likely due to the fact that when using ILNP, each sensor will employ multipath communication by using a deficit round robin algorithm to alternate which of the sink's locators it sends to, whereas when using IPv6, sensors will arbitrarily choose a source and destination locator to identify its connection to the sink and it will send to this destination locator throughout the experiment.

Therefore it is possible that the locators which the IPv6 sensors arbitrarily chose resulted in longer paths which means that the readings will traverse more hops and more network packets will be generated which will increase packet overhead. This is in contrast to the ILNP sensors which will distribute their readings equally over both the sink's locators, and be guaranteed to sometimes choose the destination locator which will result in the fewest hops.

Additionally, the arbitrary nature of how the IPv6 sensors choose which destination locator to send to throughout the experiment could explain its higher variance in packet overhead in comparison to ILNP, whose sensors will evenly distribute their readings over both destination locators.

Finally, the high packet overhead observed for both protocols is likely because unlike infrastructure networks where a single DNS call can be used to find connectivity information for a host, advertisements must regularly be flooded throughout the network in this ad-hoc emulation. This will contribute to a high packet overhead.

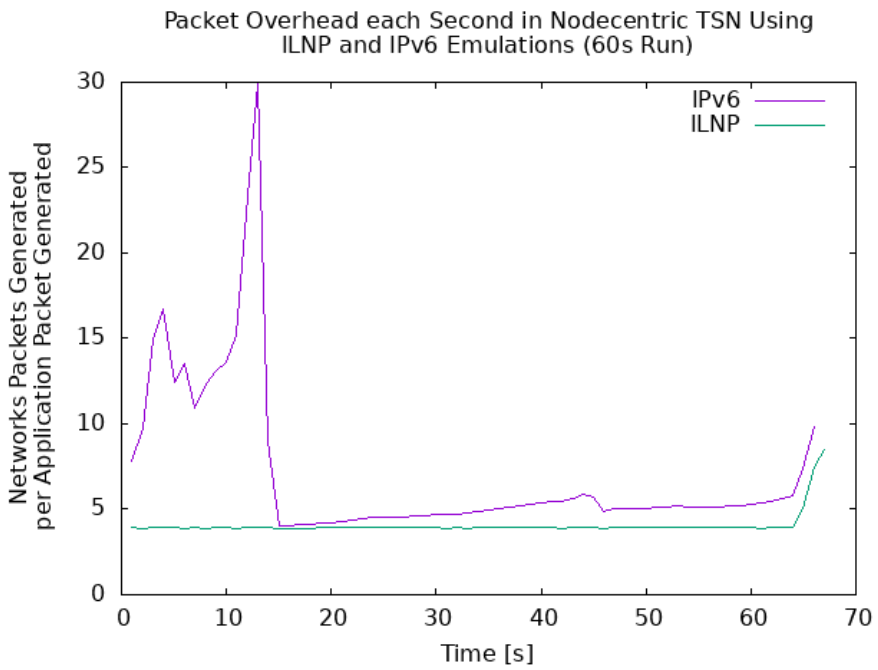

Figure 38: Graph of Packet Overhead per Second in Nodecentric TSN

Observations

The nodecentric TSN using ILNP exhibited a consistent packet overhead of 4 network packets per application packet each second throughout the selected run, whereas the selected run which used IPv6 exhibits some very high spikes in packet overhead at the start, reaching 30 network packets generated per application packet, before settling down at around 5.

Both runs exhibit a small increase in packet overhead at the end.

Analysis

The equilibrium packet overhead incurred by the nodecentric TSN when it uses IPv6 is slightly higher than when it uses ILNP which is consistent with previous analysis.

The small increase in packet overhead at the end is because the sensors stop sending data so the ratio of network packets generated per application packet generated increases.

The spikes observed at the start of the IPv6 nodecentric TSN run shown was

not expected, however the emulated IoTs were not the only processes running on their lab machines so these spikes could be due to the effects of competing with other processes or some network congestion from other processes running in the school network at that time. This pattern is not expected to replicate, and this is a good example of irregular behaviour which can show up in single runs.

The graphs which show 15 runs will reduce the influence of outlying irregular behaviour displayed in single runs, which is why the graphs displaying data from 15 runs are the ones upon which any conclusions will be based.

The graphs showing byte overhead are very similar to those showing packet overhead as each application packet is the same size, therefore to avoid repeating analysis, these graphs were moved to the "Remaining Graphs" appendix.

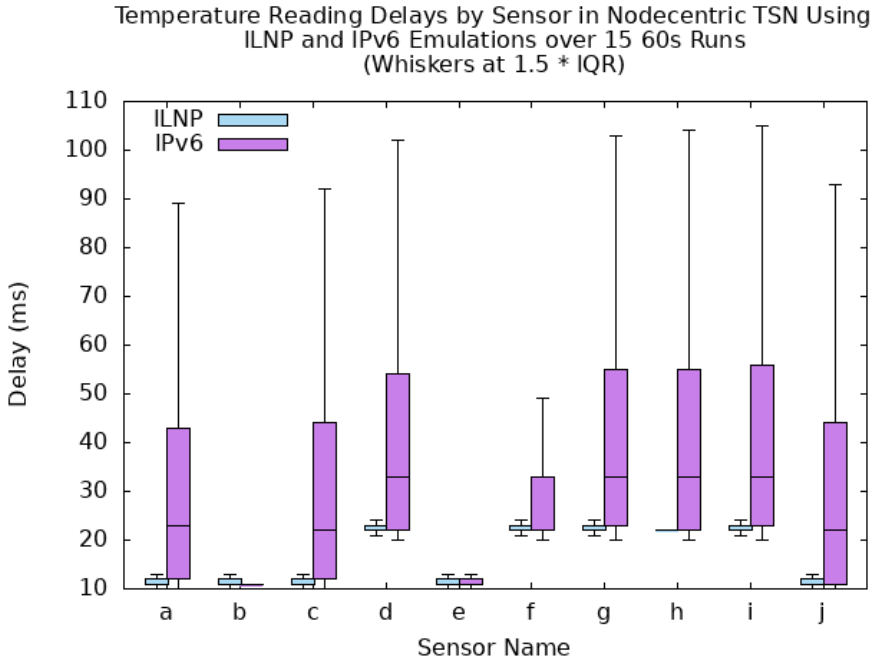

Figure 39: Graph of Temperature Reading Delays in Nodecentric TSN

Observations

When using ILNP, the reading delays from most sensors in the nodecentric TSN exhibit much less variance than when IPv6 is used, with a couple sensors showing equivalent variance in reading delay.

Additionally, when using ILNP, the reading delay from most sensors in the nodecentric TSN is much lower than when IPv6 is used, with a couple sensors showing equivalent reading delay.

Analysis

This backs up the hypothesis from previous analysis that the IPv6 mechanism of arbitrarily choosing a destination locator may result in a suboptimal path which traverses more hops being used for the entire run. This will clearly result in higher delay, and the ILNP mechanism of distributing load across the set of valid destination locators evenly is guaranteed to sometimes choose the most optimal path with fewest hops and lowest delay.

Additionally, the higher variance in reading delay for IPv6-enabled sensors is likely because the destination locator which is used for the entire run will help determine the path readings take to the sink, and it is chosen arbitrarily. ILNP's mechanism of distributing load across the set of valid destination locators evenly means that each sensor will send half its readings to each destination locator every run, hence the lower variance in reading delay for ILNP.

Finally, the higher delay in the IPv6 nodecentric TSN could also be due to poorer load distribution as it does not use multipath communication. This could result in some nodes getting congested and becoming a bottleneck while other nodes are underutilised. ILNP's propensity for multipath communication makes this situation less likely and therefore makes higher delay less likely.

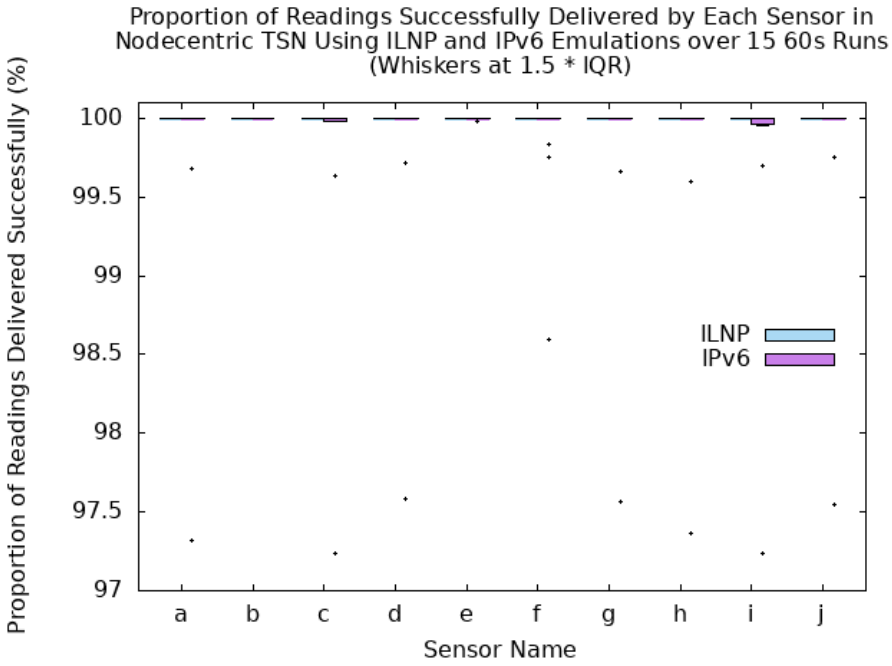

Figure 40: Graph of Temperature Reading Delivery Rates in Nodecentric TSN

Observations

This graph shows that when using ILNP, each sensor very consistently achieves a nearly perfect reading delivery rate, whereas when using IPv6, most sensors usually achieve a nearly perfect reading delivery rate, however it achieves this less consistently as it incurs more outliers with poorer reading delivery rates.

Analysis

This could due to IPv6's inability to take advantage of multipath communication resulting in poor load distribution, and some nodes being overwhelmed with packets it must forward, and these packets may then be dropped or delayed to an unacceptable degree.

Additionally, it may be due to the fact that IPv6 sensors may choose destination locators to send to for the entire run which must traverse a longer path which means that there are more routers each reading must go through, which means there are more opportunities for packet loss.

The arbitrary nature of which destination locator the IPv6 sensors choose could be the cause of the inconsistency in reading delivery rates and higher number

of outliers seen. This is in contrast to ILNP's approach of evenly distributing over every valid destination locator every run.

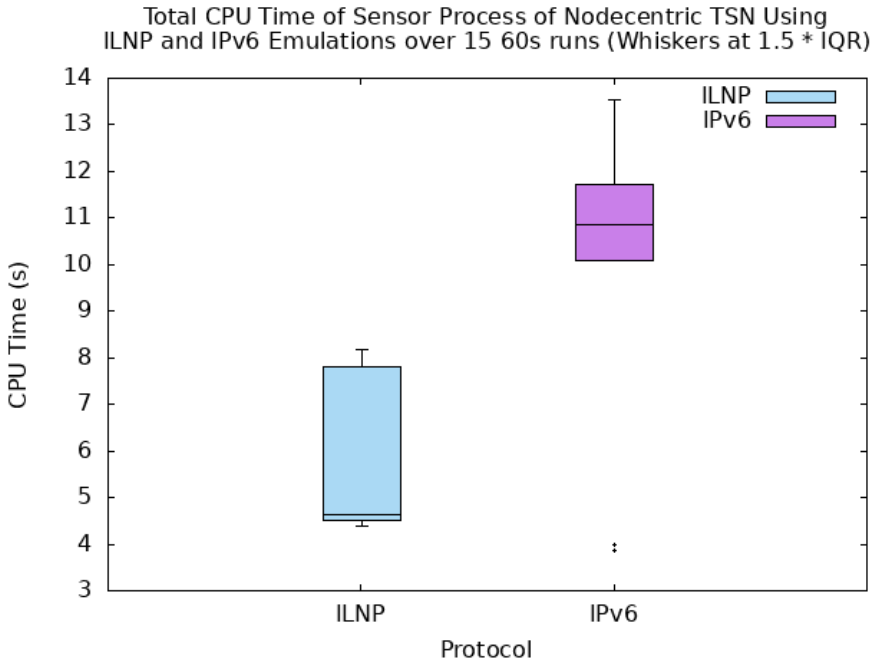

Figure 41: Graph showing Sensor F's CPU usage over 15 runs

Observations

The process hosting sensor f incurs more CPU usage over the 15 runs where IPv6 was used, than the 15 runs when ILNP was used.

Analysis

The codebases of both emulated network protocols are virtually identical, and sensor f is not mobile and therefore the code handling mobility, which does differ between the emulated network protocols, is not executed. Therefore it is more likely that this difference in CPU usage is due to architectural coincidence and the effects of other processes running on the same machine.

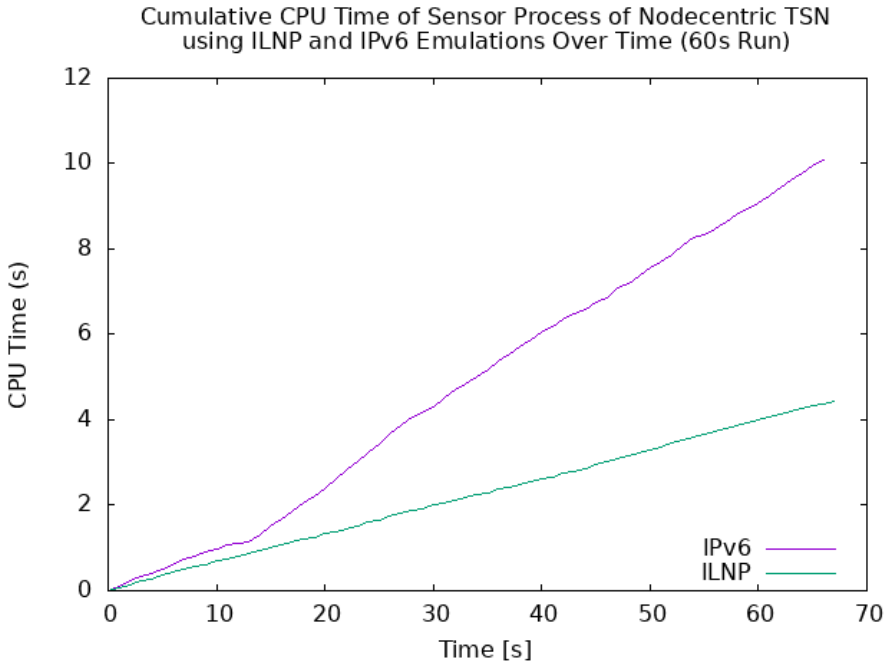

Figure 42: Graph showing Sensor F's CPU usage over a single run

Observations

The process hosting sensor f incurs more CPU usage over the run where IPv6 was used, than the run when ILNP was used.

Analysis

The codebases of both emulated network protocols are virtually identical, and sensor f is not mobile and therefore the code handling mobility, which does differ between the emulated network protocols, is not executed. Therefore it is more likely that this difference in CPU usage is due to architectural coincidence and the effects of other processes running on the same machine.

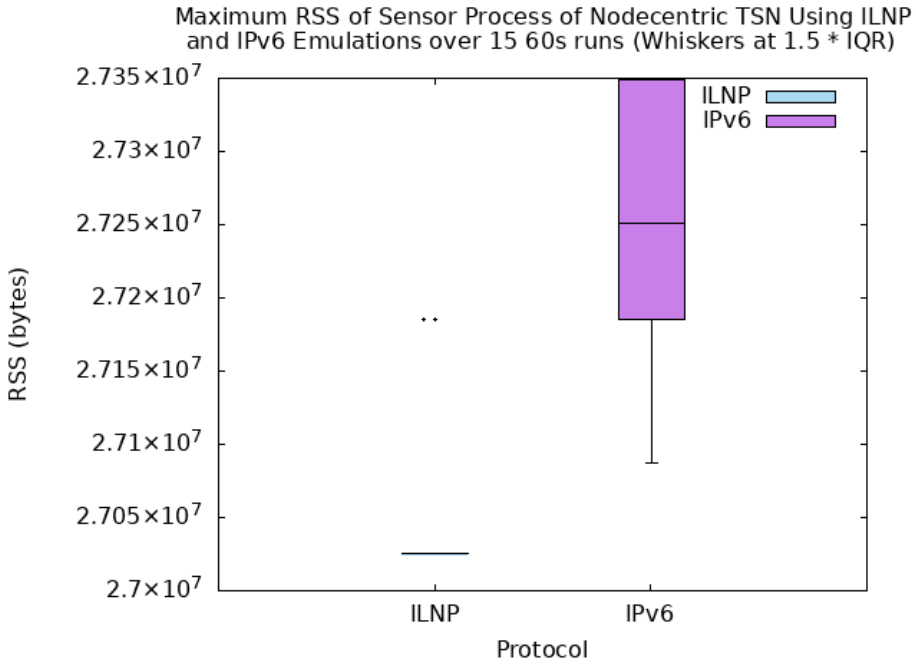

Figure 43: Graph showing Sensor F's RSS over 15 runs

Observations

The process hosting sensor f uses more physical memory over the 15 runs where IPv6 was used, than the 15 runs when ILNP was used.

Analysis

The codebases of both emulated network protocols are virtually identical, and sensor f is not mobile and therefore the code handling mobility, which does differ between the emulated network protocols, is not executed. Therefore it is more likely that this difference in physical memory size is due to architectural coincidence and the effects of other processes running on the same machine.

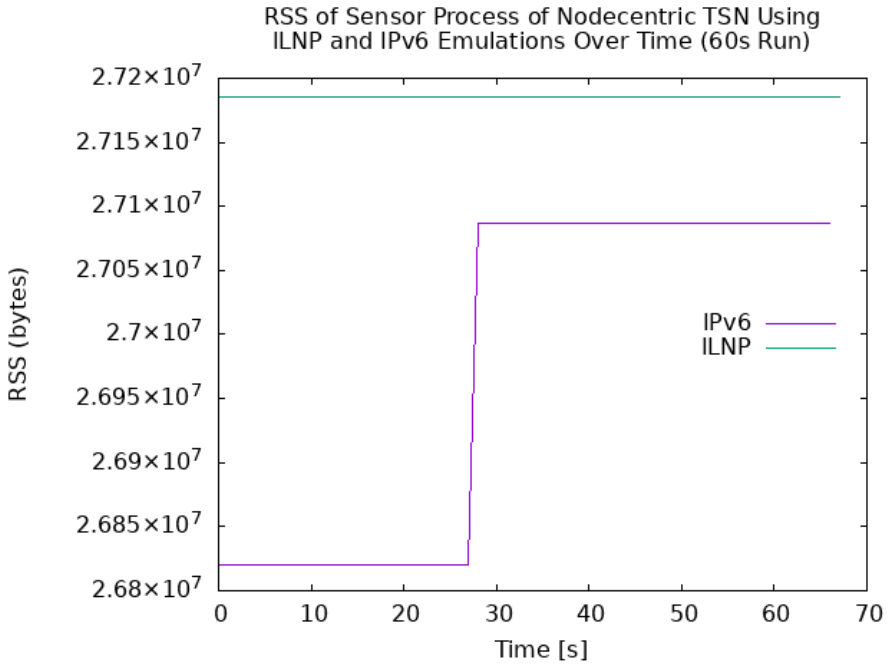

Figure 44: Graph showing Sensor F's RSS over a single run

Observations

The process hosting sensor f uses more physical memory over the run where ILNP was used, than the run when IPv6 was used. Additionally, the size of physical memory used by the process hosting sensor f sharply increases halfway through the IPv6 run.

Analysis

The codebases of both emulated network protocols are virtually identical, and sensor f is not mobile and therefore the code handling mobility, which does differ between the emulated network protocols, is not executed. Therefore it is more likely that this difference in physical memory size is due to architectural coincidence and the effects of other processes running on the same machine.

Additionally the sharp increase in physical memory halfway through the IPv6 run shown does not correspond to any notable memory intensive event, which further demonstrates that the physical memory size reported is strongly influenced by architectural coincidence and the effects of other processes running on the same machine.

11.4 Data-Centric Temperature Sensor Network

11.4.1 Description

The second scenario for which an experimental scenario application will be developed using the API is another temperature sensor network, however this time a data-centric approach is used[35]. This means instead of sensors pushing their readings at regular intervals, sinks broadcast their interest in certain types of readings at regular intervals, and if a sensor obtains a reading satisfying those conditions, it sends it to the interested sink. In this implementation, the sink broadcasts interest in temperature readings between 0° and 49° inclusive, and the sensors alternate between generating a reading by obtaining a random integer within this range which will be sent to the sink, and one which will not be sent as it does not fall within the range.

Like the nodecentric temperature sensor network, this application has potential for multipath communication, but not mobility. The initial topology for this application is the same as the one used in the nodecentric temperature sensor network as shown in figure 35.

11.4.2 Configuration

This application uses the same application configuration parameters as the nodecentric temperature sensor network, and the following two additional parameters.

```
gradient_timeout: 1
gradient_interval: 0.1
```

Their semantics are as follows:

- `gradient_timeout` is the length of time for which a sensor retains an advertised gradient.
- `gradient_interval` is the time period between which successive interest datagrams are flooded by the sink.

11.4.3 Implementation

The function of this scenario is very similar to that of the previous scenario, and a modular design allowed for the majority of the code for the nodecentric temperature sensor network to be reused for the datacentric temperature sensor network.

However unlike the previous application, the sink in this application has a task which floods a datagram which registers its interest in a specific temperature

range every `gradient_interval`. Sensors record this interest (called a gradient) and when they generate a reading, they only send to a sink, if there is a gradient established in the previous `gradient_timeout` which states they are interested in it.

The structure of the packet which sinks use to broadcast their interest in a temperature range is shown below. The sequence number is included so old expressions of interest are ignored.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
Type	Lower Bound	Upper Bound	Unused
Sequence number			

Figure 45: Structure of packet used by the sink to express interest in a temperature range

11.4.4 Method

The metrics being measured in this application are the same as the ones that were measured in the nodecentric temperature sensor network application, and the values are obtained in the same way too (however the paradigm command line argument for the `run_experiment` script will be set to "datacentric" instead of "nodecentric").

Note that since the application layer datagrams which the sink floods to register its interest in a temperature range do not reach the sink, they are not counted in the delay or delivery rate metrics. They are however, still included in the overhead statistics.

11.4.5 Results

This section presents the graphs of the data which were created in accordance with the steps outlined in the statistical handling section, along with associated observations and analysis.

Some of the graphs were omitted from this section and moved to the "Remaining Graphs" appendix to prevent reiteration of the previous observations and analysis. The omitted graphs include the byte overhead graphs as these are very similar to the packet overhead graphs.

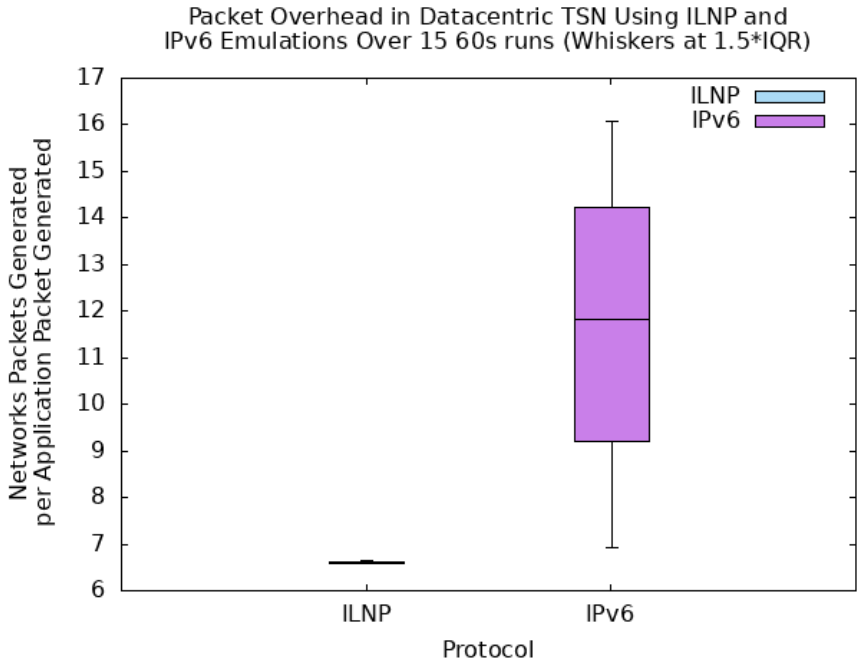

Figure 46: Graph of Packet Overhead in Datacentric TSN Over 15 Runs

Observations

The observations that the usage of IPv6 results in a higher packet overhead, and a higher variance in packet overhead than the usage of ILNP for the nodecentric TSN application, can also be made about this datacentric TSN application. The difference is that the datacentric TSN application has higher packet overheads over each protocol than its nodecentric counterparts.

Analysis

The analysis that the IPv6 sensors' inability to use multipath communication is responsible for the higher packet overhead and higher variance in packet overhead in the nodecentric TSN also applies to this datacentric TSN application.

The reason that the datacentric TSN application has higher packet overheads over each protocol than its nodecentric counterparts is because half as many application packets get sent as only temperature readings that the sink is interested in get sent.

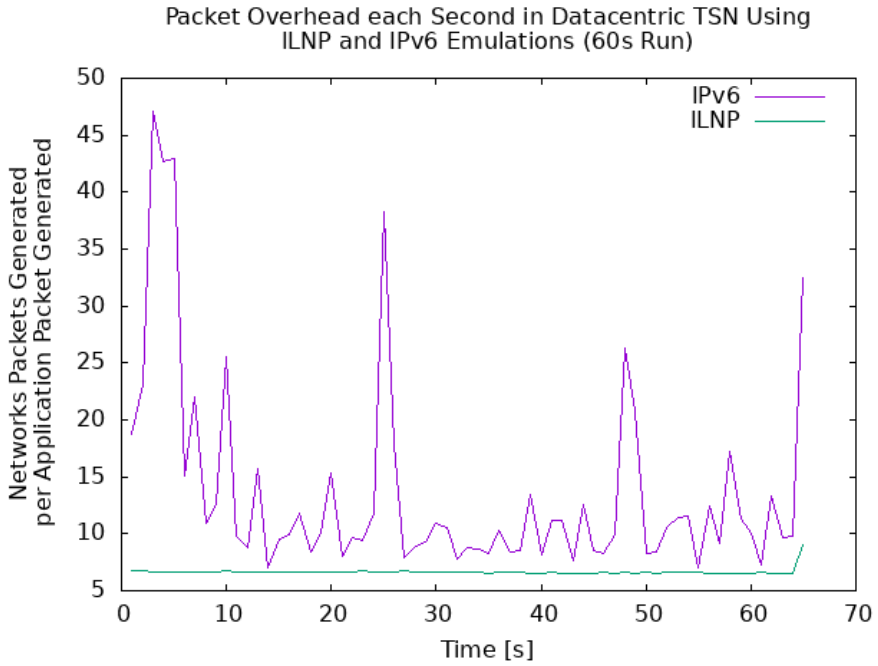

Figure 47: Graph of Packet Overhead per Second in Datacentric TSN

Observations

Packet overhead is higher for the IPv6 run than the ILNP run, and packet overhead is a lot more stable for ILNP than IPv6 during these runs.

Analysis

The fact that packet overhead is higher for IPv6 than ILNP is to be expected given the previous analysis, however the extreme fluctuations in the packet overhead for the IPv6 run was not expected. No other characteristic of datacentric paradigms or IPv6 should result in a pattern like this, so this is likely irregular behaviour resulting from the effects of other processes on the same computers.

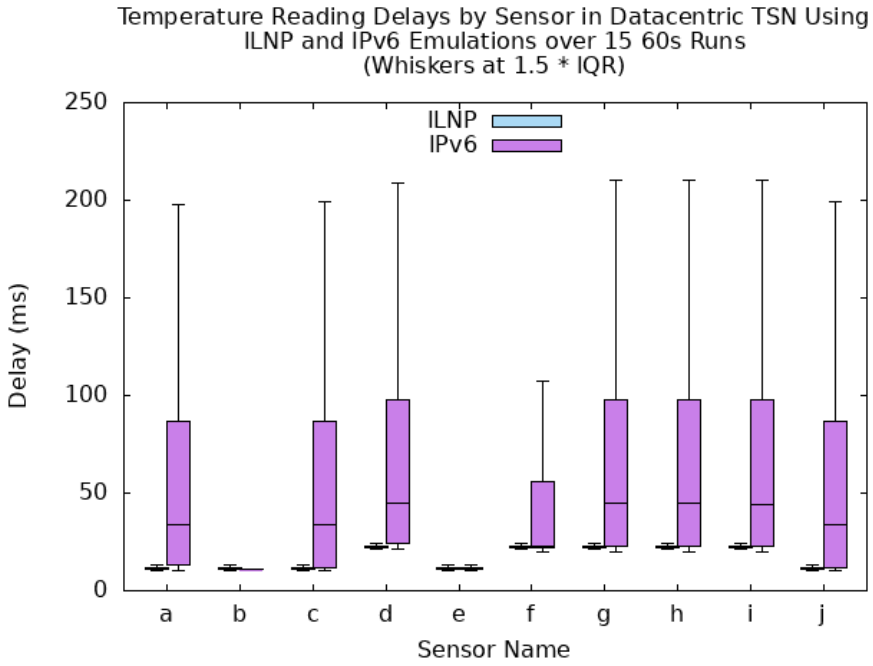

Figure 48: Graph of Temperature Reading Delays in Datacentric TSN

Observations

As with the nodecentric TSN, when using ILNP, the reading delays from most sensors in the datacentric TSN exhibit much less variance than when IPv6 is used, with a couple sensors showing equivalent variance in reading delay.

Additionally, when using ILNP, the reading delay from most sensors in the datacentric TSN is much lower than when IPv6 is used, with a couple of sensors showing equivalent reading delay.

Finally, the delays observed in the datacentric TSN are larger than those observed in the nodecentric TSN.

Analysis

The analysis that the increased reading delay, and the increased variance in reading delay observed in IPv6 sensors in the nodecentric TSN is due to their inability to use multipath communication, and the resultant poor load balancing and potentially suboptimal routes taken also applies to the datacentric model.

Additionally, the fact that the delays are higher in the datacentric TSN than the

nodecentric TSN is due to the fact that the sink floods datagrams which register its interest in certain temperature ranges, and this results in higher congestion in the network.

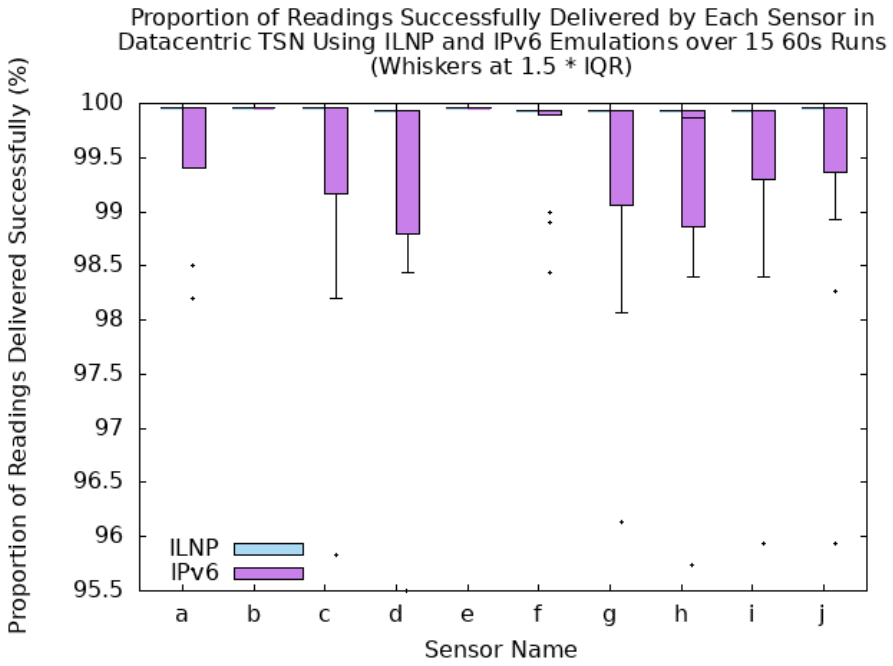

Figure 49: Graph of Temperature Reading Delivery Rates in Datacentric TSN

Observations

Again, as in the nodecentric TSN, when using ILNP the delivery rate is consistently near perfect, and when using IPv6 it is still very good, however there is greater variance and more outliers in which the delivery rate falls short of near perfect. These outliers have a lower delivery rate than the IPv6 outliers observed in the nodecentric TSN.

Analysis

The analysis that the lower delivery rates and higher variance in delivery rate for the nodecentric TSN when it uses IPv6 is due to IPv6's inability to use multipath communication, and the resultant poor load balancing and potentially suboptimal routes taken also applies to this datacentric application.

Additionally, the outliers for the IPv6 sensors have a lower delivery rate than

the corresponding outliers in the nodecentric model as the sinks flooding of datagrams which register its interest in certain temperature ranges causes more congestion, and therefore more nodes to become overwhelmed and drop packets.

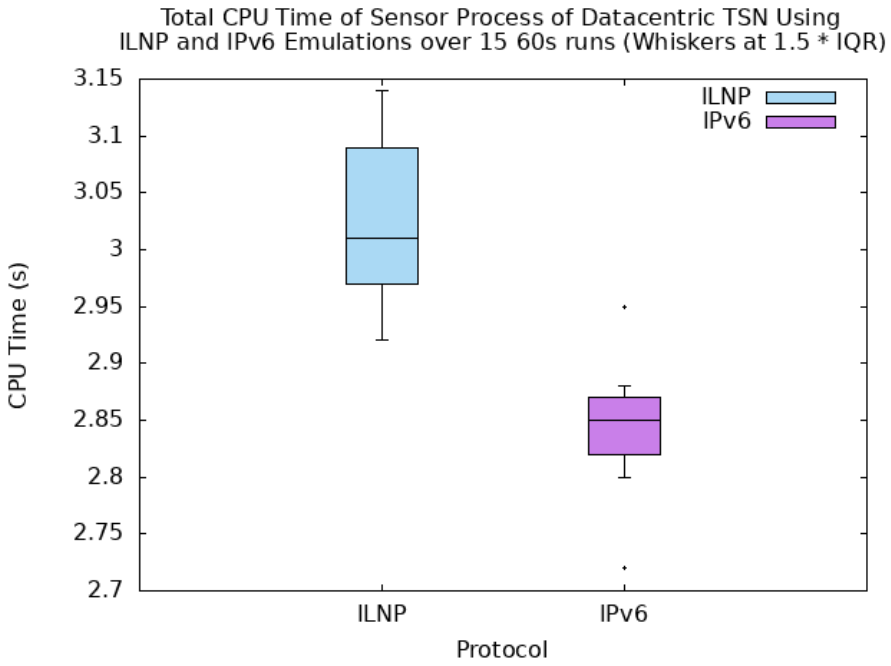

Figure 50: Graph showing Sensor F's CPU usage over 15 runs

Observations

The process hosting sensor f incurs more CPU usage over the 15 runs where ILNP was used, than the 15 runs when IPv6 was used.

Analysis

The codebases of both emulated network protocols are virtually identical, and sensor f is not mobile and therefore the code handling mobility, which does differ between the emulated network protocols, is not executed. Additionally, the codebase of the nodecentric TSN is very similar to that of the datacentric TSN, however the comparison between sensor f's CPU usage when using the different emulated network protocols yields a different results in the nodecentric and datacentric experiments. Therefore it is more likely that this difference in CPU usage is due to architectural coincidence and the effects of other processes running on the same machine.

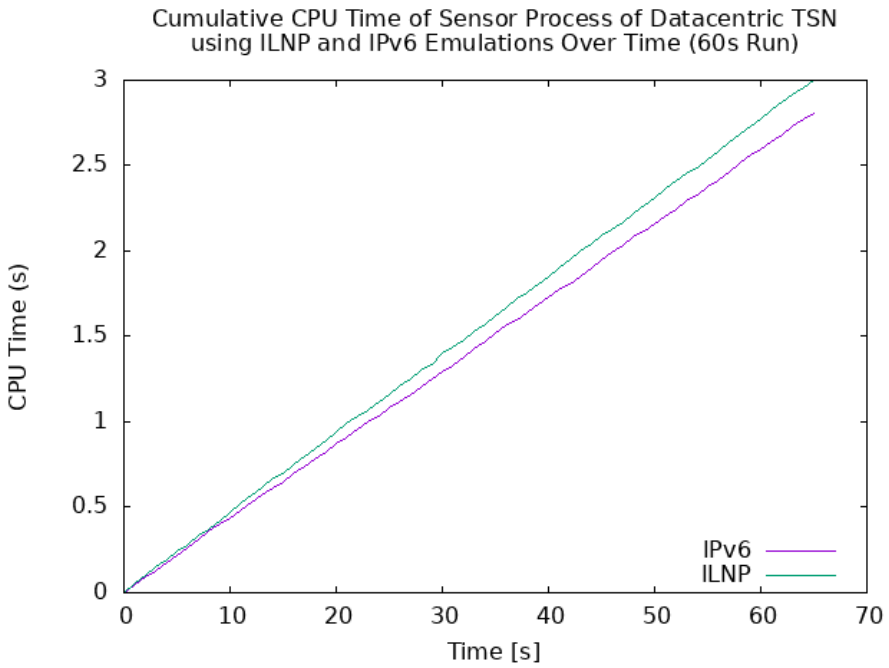

Figure 51: Graph showing Sensor F's CPU usage over a single run

Observations

The process hosting sensor f incurs slightly more CPU usage over the run where ILNP was used, than the run when IPv6 was used.

Analysis

See analysis of the graph in figure 50.

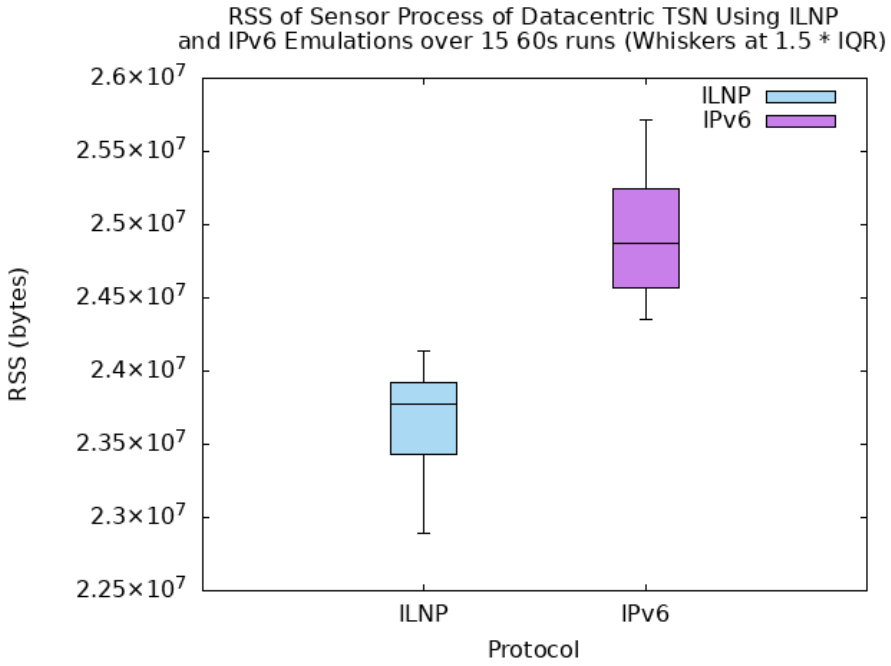

Figure 52: Graph showing Sensor F's RSS over 15 runs

Observations

The process hosting sensor f uses more physical memory over the 15 runs where IPv6 was used, than the 15 runs when ILNP was used.

Analysis

The codebases of both emulated network protocols are virtually identical, and sensor f is not mobile and therefore the code handling mobility, which does differ between the emulated network protocols, is not executed. Therefore it is more likely that this difference in physical memory size is due to architectural coincidence and the effects of other processes running on the same machine.

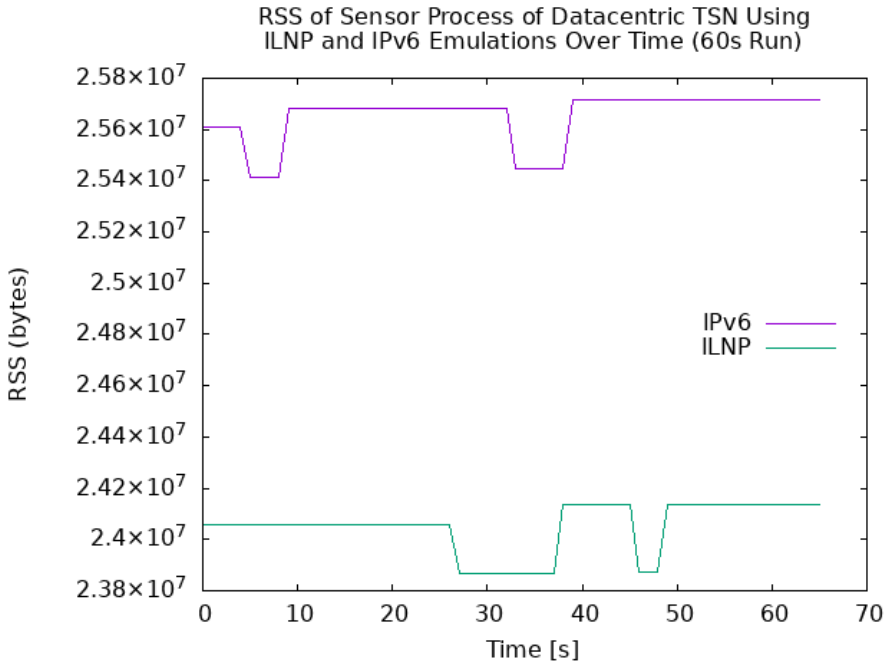

Figure 53: Graph showing Sensor F's RSS over a single run

Observations

The process hosting sensor f uses more physical memory over the run where IPv6 was used, than the run when ILNP was used. Additionally, the size of physical memory varies in a manner which does not seem to correlate with any memory-intensive event of the application.

Analysis

See analysis of graph in figure 44.

11.5 Animal Tracker

11.5.1 Description

The third experimental scenario application is an animal tracker which consists of a sink which makes regular queries about the location of an animal by sending it unicast tracking requests. This animal will move between several stationary routers which each have just one invariant locator, and the animal's IoT tracker will associate with the invariant locator of the router it is closest to. Therefore once the animal receives a tracking request, it can observe the link layer datagram to observe the locator it was reached on, use this to deduce its location,

and it will reply to the sink with this information.

The toplogy diagram in figure 54 which shows the initial topology for the animal tracker application shows that from every router there are multiple paths to the sink, therefore there is potential for multipath communication in this application. The emulation of ILNP-enabled IoT devices exploits this as the send primitive provided by the API for ILNP nodes uses a deficit round robin algorithm to distribute the load across the sink's 2 locators.

Mobility is demonstrated in this application too as the animal node regularly changes connectivity by moving location and associating with the locator of a different stationary router.

The initial topology of the emulated IoT nodes in the animal tracker application is shown in the diagram below.

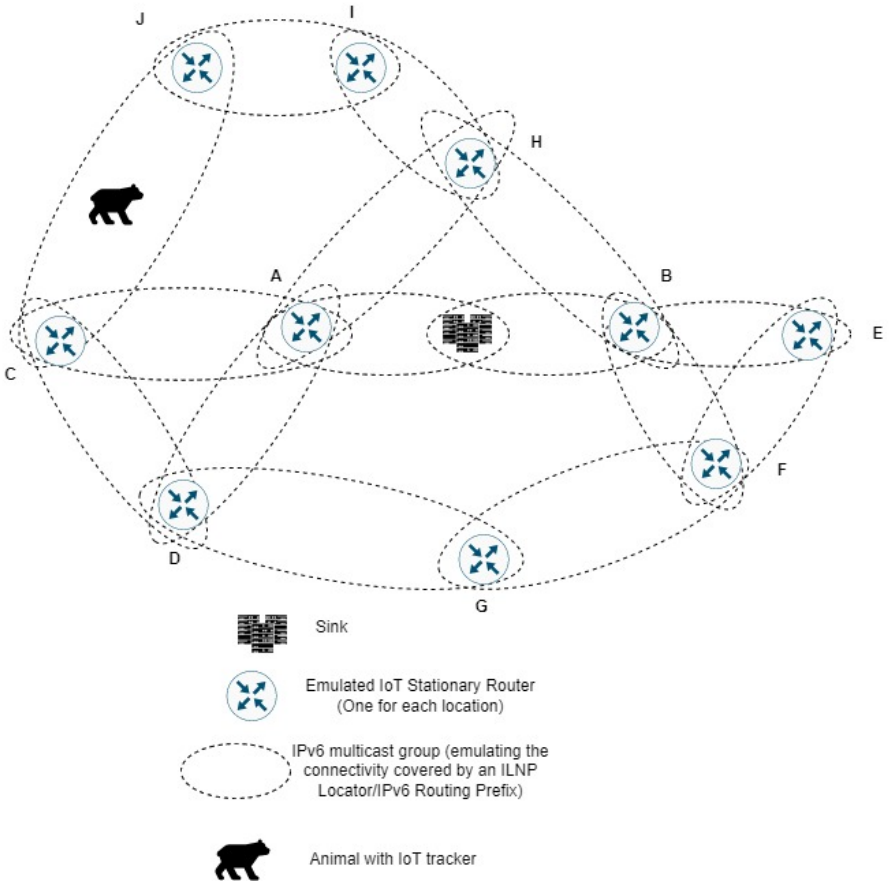

Figure 54: Initial topology for the animal tracker application

Note that the animal IoT is non-routing.

11.5.2 Configuration

The animal tracker application is run with the following application configuration parameters:

```
app_port: 3000
server_port: 34355
emulation_port: 21878
reporting_interval: 1
sink_query_interval: 0.001
handshake_timeout: 2
hosts_file: ../../../hostnames
topology_file: topology.yml
```

```
queries_to_send: 60000
mobility_interval: 0.3
final_wait: 0.05
```

The semantics of the fields most relevant to the operation of the application are as follows:

- `reporting_interval` is the number of seconds between each object's requests for resource and overhead statistics from their emulated IoT device.
- `sink_query_interval` is the time period between which successive unicast tracking requests are sent by the sink to the animal.
- `queries_to_send` is the number of queries the sink will send.
- `mobility_interval` is the interval between which the animal will make successive changes in connectivity meant to simulate movement.

11.5.3 Implementation

The identifier of the animal is obtained after its IoT device is set up using the `get_id` API call, and this identifier is passed to the sink so it may send its unicast tracking requests to the animal by passing the identifier to the send API call.

The tracking request simply includes a type field, and a request number. Once a tracking request is sent, the request number is stored with the time the request was sent. This means that outstanding requests can be kept track of, and once a request is responded to, the time it took for a response to be received can be calculated. The packet structure of the tracking request is shown below.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Type	Unused
Request Number	

Figure 55: Packet structure of tracking request

Similarly, the reply includes a type field and the request number of the request it is responding to. Additionally, the locator it was reached on is included as an indicator of the animal's location. The animal's IoT obtains this from the locator of the link frame of the request.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Type	Unused
Request Number	
Locator	
Locator Continued	

Figure 56: Packet structure of tracking response

The locator which the animal's emulated IoT is associated with is conceptually the location of the animal. The movement of the animal is simulated by the animal object having an asynchronous task which chooses the invariant locator of a random, different stationary router every `mobility_interval`, and it associates with it using the `replace` API call.

11.5.4 Method

In this application, the following metrics are recorded:

- CPU Usage of the process running the animal's emulated IoT device.
- The resident set size of the process running the animal's emulated IoT device.
- The packet overhead throughout the emulated network.
- The byte overhead throughout the emulated network.
- Latency of responses of requests for the animal's location.
- Proportion of successful requests for the animal's location

This is achieved by running the `run_experiment.py` script in the `src/experiments/animal_tracker` folder which sets up the emulated IoT devices and arranges them in the initial topology as specified earlier, then initialises the Animal, Router and Sink objects. Its usage is as follows:

```
./run_experiment <app_config_file> <protocol> <no_of_runs>
```

where the protocol is either "ilnp" or "ipv6".

Once all this has been completed, the application may begin. The sink will make `queries_to_send` unicast requests to the animal for its location by sending a tracking request datagram to the animal IoT every `sink_query_interval`.

The animal will respond to every request it sees and for each response that gets back to the sink, the sink will record that the request with the given sequence number is resolved, and record the time it took to resolve by taking the time the request was resolved away from the time it was made. This can be used to generate graphs which show the tracking response latency, and tracking query success rate as specified in the statistical handling section.

Additionally, each object has an asynchronous task which reports overhead statistics and resource usage statistics every `reporting_interval` which can be used to generate graphs on resource usage and overhead as specified in the statistical handling section. The animal IoT was chosen as the node which has its resource usage graphed as it performs mobility, and it would be interesting to see the impact of this on resource consumption.

11.5.5 Results

This section presents the graphs of the data which were created in accordance with the steps outlined in the statistical handling section, along with associated observations and analysis.

Some of the graphs were omitted from this section and moved to the "Remaining Graphs" appendix to prevent reiteration of previous observations and analysis. The omitted graphs include the byte overhead graphs, as they are very similar to the packet overhead ones, and the CPU and RSS ones as previous experiments have suggested that the results in those graphs are too heavily influenced by the effects of other processes on the same machines and architectural coincidence to make any meaningful conclusions about the effect of the emulated network protocols.

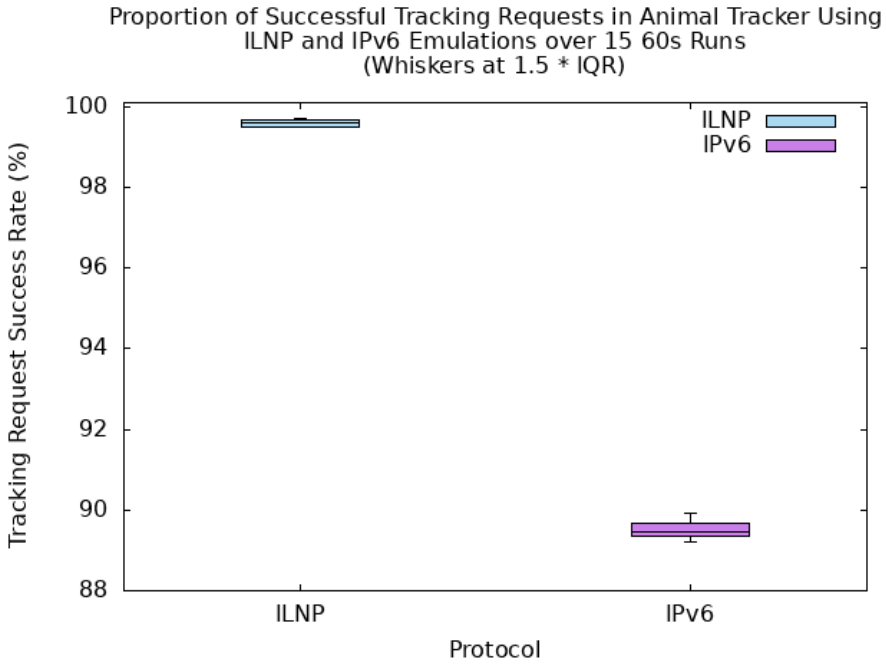

Figure 57: Graph of Query Success Rate of Animal Tracker Over 15 Runs

Observations

This graph shows that the animal tracker has a much higher tracking query success rate when it uses ILNP than when it uses IPv6.

Analysis

This is because when ILNP is used, the animal IoT uses soft handoff which maintains the previous connectivity until the updated connectivity is acknowledged by all of its correspondents, meaning that it will be able to receive and respond to queries which are sent to the old locator while the sink is unaware that the animal IoT has changed connectivity. IPv6 on the other hand leaves the old locator without an acknowledgement from its correspondents, so the sink may send data to the previous locator despite the animal not being associated with it anymore, resulting in a failed query.

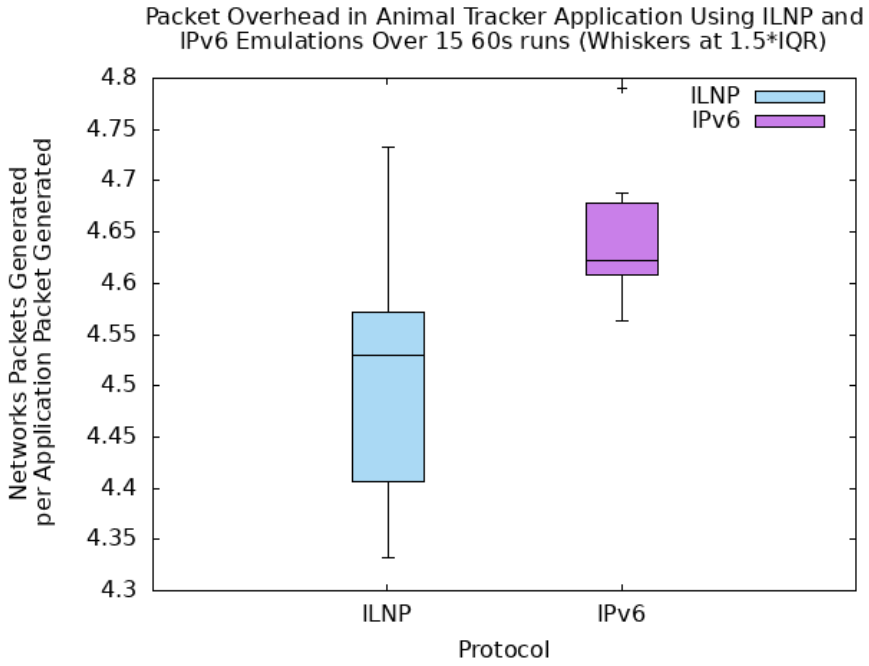

Figure 58: Graph of Packet Overhead in Animal Tracker Over 15 Runs

Observations

This graph shows that the animal tracker incurs a lower median packet overhead when it uses ILNP, than when it uses IPv6. However it is not a huge difference.

Analysis

There are two opposing factors here. The first is that when using ILNP, more network layer packets will be sent, as the locator updates and acknowledgements associated with soft handoff are being sent. However, more application packets are sent when using ILNP too, as the usage of soft handoff means more requests get to the animal IoT, and therefore more responses are generated. The difference in the number of application packets generated clearly has a greater effect than the difference in the number of network packets generated, so the usage of ILNP results in a lower median packet overhead in this animal tracker application.

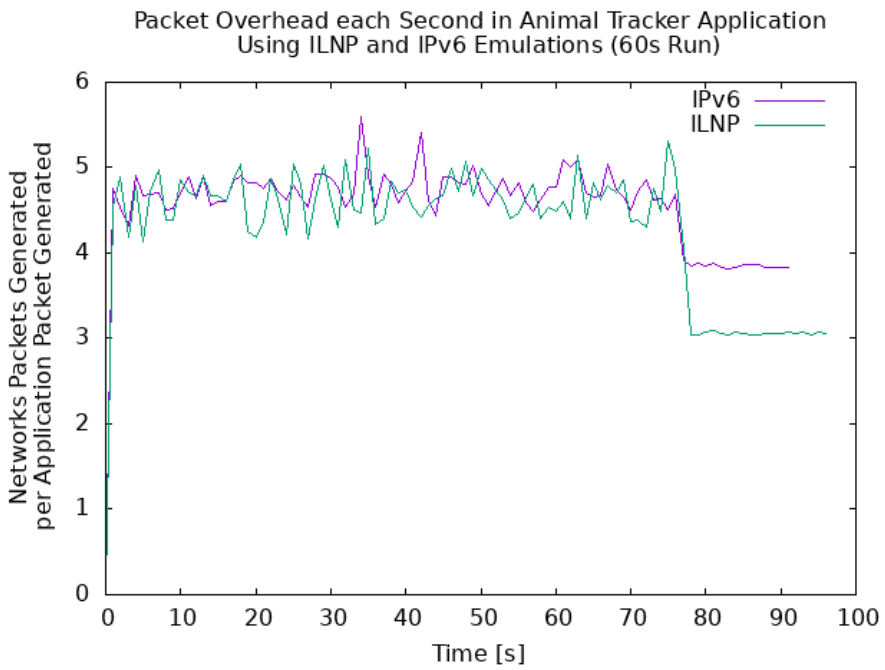

Figure 59: Graph of Packet Overhead per Second in Animal Tracker

Observations

The packet overhead in both runs of the animal tracker fluctuates a lot however the packet overhead when using ILNP appears to be slightly lower overall.

The fluctuations appear to stop and packet overhead stabilises for both runs at around 70 seconds.

Analysis

The fluctuations in packet overhead is probably due to the mobility of the animal, so the paths that the tracking requests and responses take are constantly changing. This means the number of hops, and therefore the number of network packets being generated are constantly changing.

The fluctuations end and packet overhead stabilises because the animal stops moving after a minute.

The fact that ILNP appears to result in less packet overhead for the animal tracker is consistent with previous analysis.

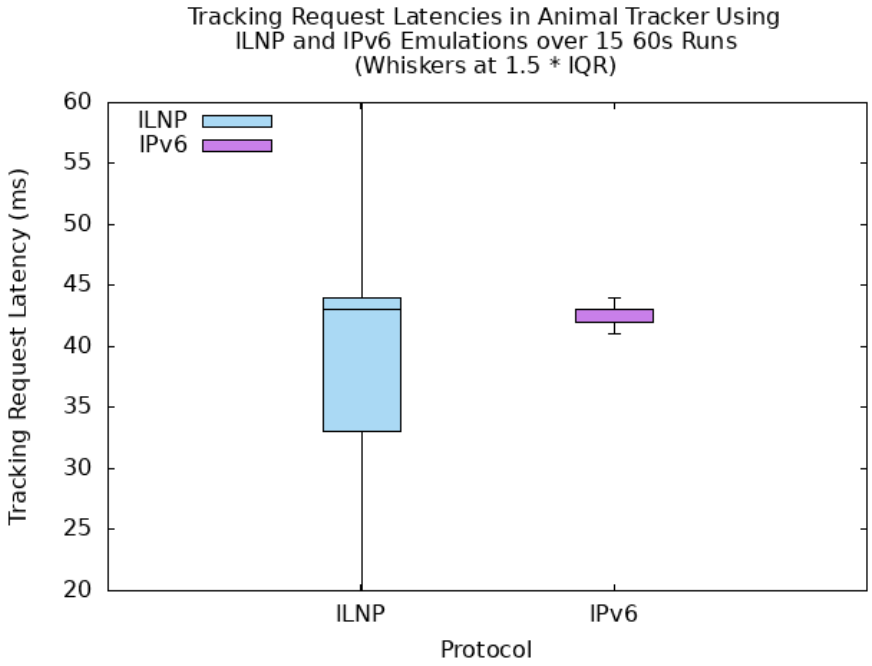

Figure 60: Graph of Tracking Request Latencies in Animal Tracker Over 15 Runs

Observations

The animal tracker reports similar median tracking request latencies for ILNP and IPv6.

Analysis

There are two opposing factors at play here. Firstly, ILNP nodes may take advantage of multipath communication which makes better load distribution more likely, and makes the scenario where a single node is overwhelmed and will incur more delay less likely. However this is partially cancelled out by the locator updates and acknowledgements which will be present in the ILNP animal tracker, which will result in some additional congestion and delay. The combination of these opposing factors results in equivalent median delay observed in the ILNP and IPv6 animal tracker applications.

11.6 Emergency Communication Network

11.6.1 Description

The final experimental scenario application is an emergency communication network which consists of a mobile sender, mobile receiver, and several mobile IoTs with one invariant locator whose only purpose is to forward packets from the sender towards the receiver. Mobiles will associate with the locators of other mobiles in such a way that every mobile can reach every other mobile, and each mobile will change the locator set it is associated with at regular intervals throughout the program to demonstrate mobility.Throughout the experiment, the sender will direct an application layer flow towards the receiver, and the impact of the mobility on this flow can be observed.

This application has potential for multipath communication as when changing the locator set, every mobile, including the sender and receiver, can join 2 locators, which will result in a topology where the sender has multiple paths to the receiver, and the sender will alternate which locator it reaches the receiver on when emulating ILNP.

This application demonstrates mobility as each node is mobile and regularly changes the locator set with which it is associated.

A diagram of the initial topology of this scenario is shown below.

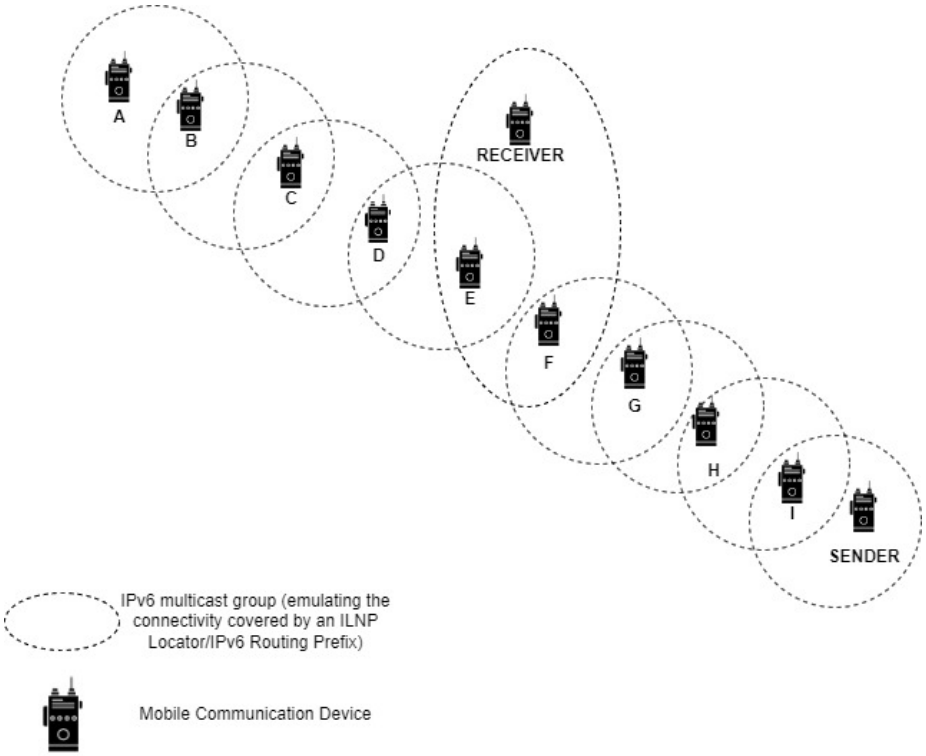

Figure 61: Initial Topology of Emergency Communication Network Experiment

Note that the sender and receiver IoTs are non-routing.

11.6.2 Configuration

The emergency communication network is run with the following application configuration:

```
app_port: 3000
server_port: 34355
emulation_port: 21878
reporting_interval: 1
flow_rate: 1000
handshake_timeout: 2
hosts_file: ../../../hostnames
dgrams_to_send: 60000
mobility_interval: 0.5
final_wait: 0.05
topology_file: topology.yml
```

The semantics of the fields most relevant to the operation of this application are as follows:

- `reporting_interval` is the number of seconds between each object's requests for resource and overhead statistics from their emulated IoT device.
- `flow_rate` is the number of application packets the sender sends to the receiver per second.
- `dgrams_to_send` is the number of packets the sender will send in the application layer flow.
- `mobility_interval` is the interval between which the each mobile will make successive changes in their locator set, in order to simulate movement.

11.6.3 Implementation

After the receiver device has its emulated IoT device set up, its identifier can be obtained using the `get_API` method. This is then passed to the sender device which in turn passes the identifier to the send API call so it can send `flow_rate` datagrams to the receiver every second.

The flow datagrams contain a sequence number, and the receiver can use these to keep track of which datagrams it received, and therefore which flow datagrams were lost too. Additionally, a timestamp is contained so the receiver can calculate the one-way delay for each flow datagram.

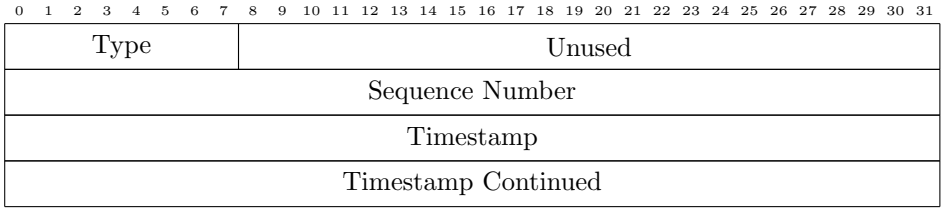

Figure 62: Packet structure of packets which make up the application layer flow

The mobile devices each simulate mobility by having an asynchronous task which changes the locator set it is associated with every `mobility_interval`. This is achieved by using the `replace` API call which replaces the set of locators the node is associated with.

To ensure that every node can connect to every other node, a strict ordering of the nodes is defined in the topology YAML file (this is simply the order they appear in this file), and each routing node will choose 2 locators from the list of invariant locators of nodes which come after it to associate with, which means all

of the routing nodes will be connected to each other. The sender and receiver do not have any invariant locators, and they choose 2 locators of any of the routing nodes to connect to, thereby benefitting from the complete connectivity of the network of routing nodes.

11.6.4 Method

In this application, the following metrics are recorded:

- CPU Usage of the process running the sender's emulated IoT device.
- The resident set size of the process running the sender's emulated IoT device.
- The packet overhead throughout the emulated network.
- The byte overhead throughout the emulated network.
- Delay of the application flow datagrams.
- Delivery rate of the application flow datagrams.

This is achieved by running the `run_experiment.py` script in the `src/experiments/emergency_comms` folder which sets up the emulated IoT devices and arranges them in the initial topology as specified earlier, then initialises the Mobile objects. Its usage is as follows:

```
./run_experiment <app_config_file> <protocol> <no_of_runs>
```

where the protocol is either "ilnp" or "ipv6".

Once all this has been set up, the application may begin. The sender will send `dgrams_to_send` flow datagrams to the receiver by sending it `flow_rate` datagrams per second. The receiver will record the sequence numbers of datagrams received and the one-way delay it incurred by subtracting the timestamp away from the time it was received. It can then generate graphs showing application layer delivery rate and delay as specified in the statistical handling section.

Finally, each object has an asynchronous task which reports overhead statistics and resource usage statistics every `reporting_interval` which can be used to generate graphs on resource usage and overhead as specified in the statistical handling section. The sender was chosen as the node which has its resource usage graphed as it is the only node that takes part in application layer sending, and it would be interesting to observe the impact of this on resource consumption.

11.6.5 Results

This section presents the graphs of the data which were created in accordance with the steps outlined in the statistical handling section, along with associated observations and analysis.

Some of the graphs were omitted from this section and moved to the "Remaining Graphs" appendix to prevent reiteration of previous observations and analysis. The omitted graphs include the byte overhead graphs, as they are very similar to the packet overhead ones, and the CPU and RSS ones as previous experiments have suggested that the results in those graphs are too heavily influenced by the effects of other processes on the same machines and architectural coincidence to make any meaningful conclusions about the effect of the emulated network protocols.

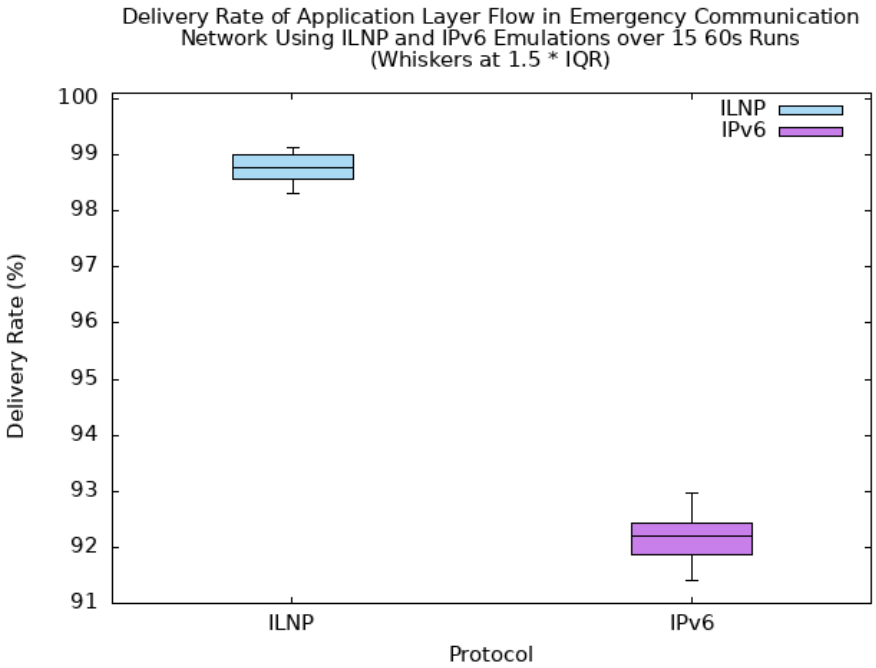

Figure 63: Graph of Application Layer Delivery Rate in Emergency Communication Network Over 15 Runs

Observations

This graphs shows that the application layer flow from the sender mobile to the receiver mobile has a much higher delivery rate when ILNP is used as opposed to IPv6.

Analysis

This is because when ILNP is used, the mobile IoTs use soft handoff which maintains the previous connectivity until the updated connectivity is acknowledged by all of its correspondents. This means that the receiver will be able to receive application packets which are sent to the old locator set while the sender is unaware that the receiver's locator set has changed and is still sending to the old locator set. When IPv6 is used on the other hand, the receiver leaves the old locator set without an acknowledgement from its correspondents, so sender will send data to the previous locator set despite the receiver not being associated with it anymore resulting in lost application packets until the next neighbour advertisement is received.

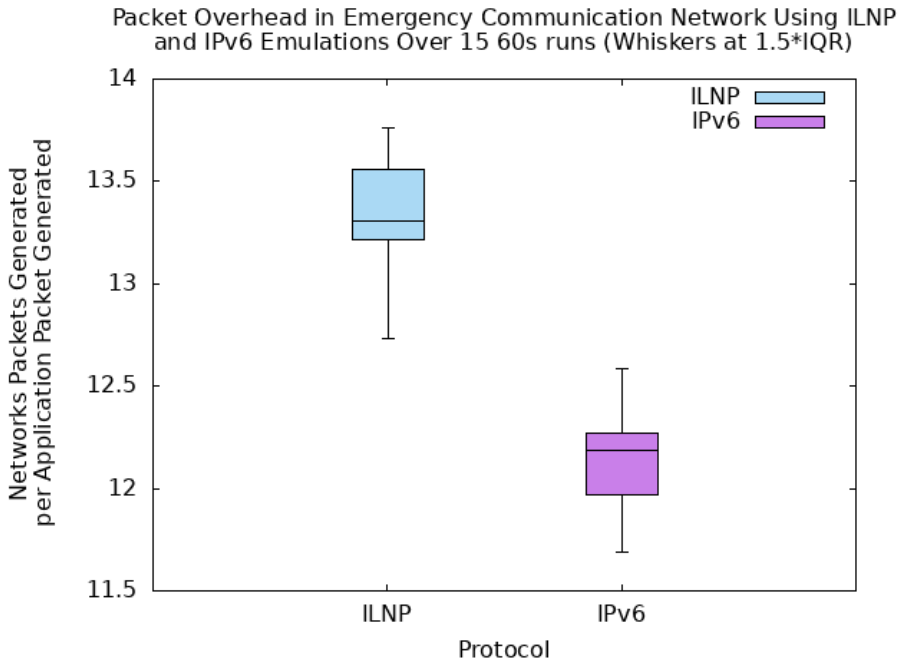

Figure 64: Graph of Packet Overhead in Emergency Communication Network Over 15 Runs

Observations

This graph shows that the emergency communication network incurs more packet overhead when ILNP is used as opposed to IPv6.

Analysis

ILNP incurs more packet overhead in this application despite its superior delivery rate because every single node sends locator updates to all its correspondents every time it moves, and the correspondents respond with acknowledgements. These locator updates are required to perform the soft handoff that results in the superior delivery rate however.

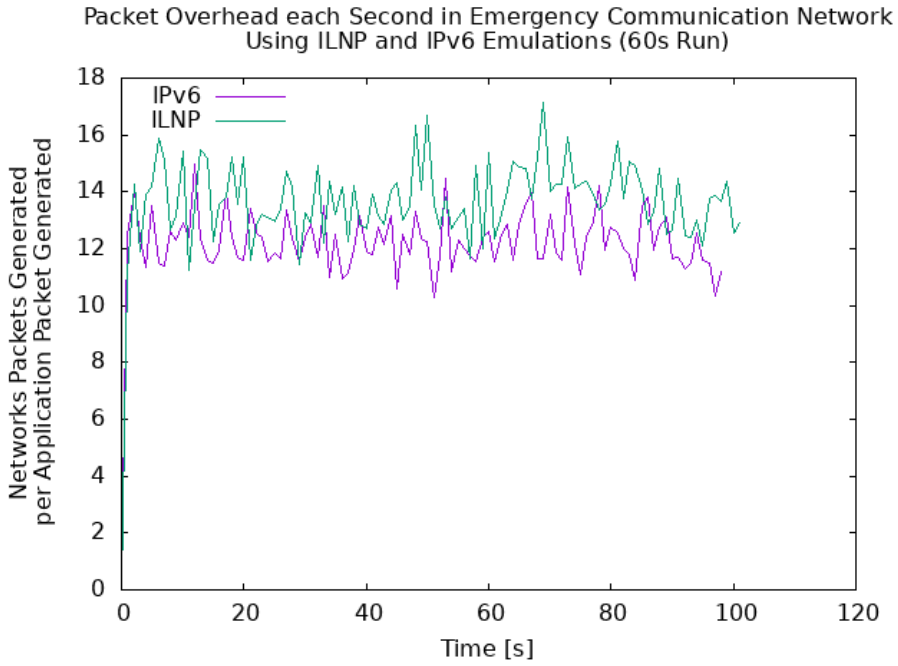

Figure 65: Graph of Packet Overhead in Emergency Communication Network per Second

Observations

In these runs of the emergency communication network, the run which uses ILNP appears to incur more packet overhead overall. Both runs exhibit a lot of fluctuation in packet overhead.

Analysis

The fluctuations in packet overhead observed are due to the fact that all the mobile nodes regularly change connectivity, which means the paths that the application packets take regularly change. This means the number of hops the

application packets traverse regularly changes, and therefore the number of network packets generated in forwarding regularly changes.

See the analysis of the graph in figure 64 for the reasons behind ILNP's higher packet overhead.

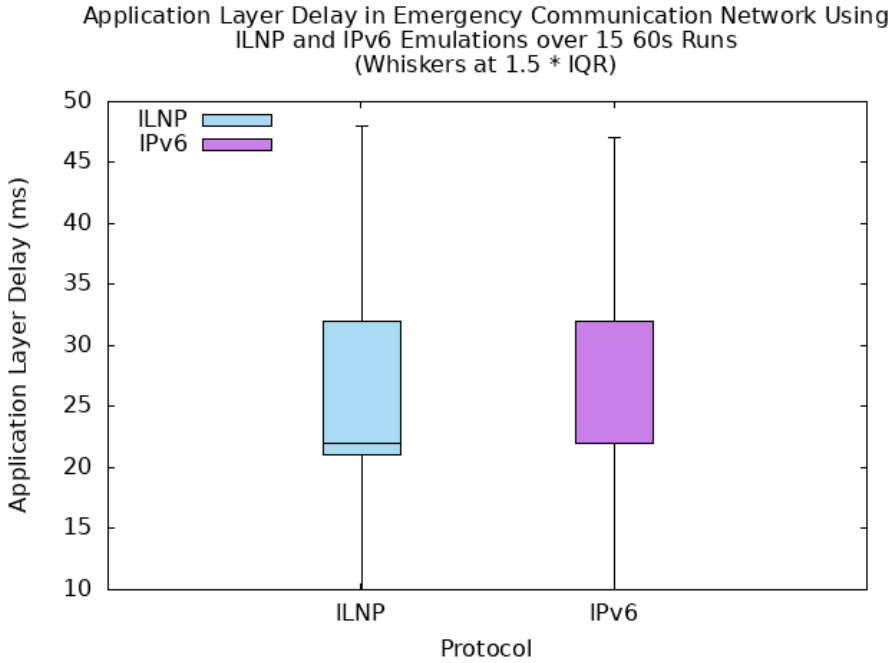

Figure 66: Graph of Application Layer Delay in Emergency Communication Network Over 15 Runs

Observations

No difference in the delay incurred by the application packets in the emergency communication network is observed between the two network layer protocols.

Analysis

There are two opposing factors at play here. Firstly, ILNP nodes may take advantage of multipath communication which makes better load distribution more likely, and makes the scenario where a single node is overwhelmed and will incur more delay less likely. However this is partially cancelled out by the added congestion in the network which results from the locator updates that the ILNP nodes send, which will result in some additional delay. The combination of these opposing factors results in equivalent median delay observed in

the ILNP and IPv6 emergency communication networks.

12 Evaluation and Conclusions

12.1 Experimental Conclusion

A common theme throughout these experimental results is that when a given experiment uses ILNP, its delivery rate metric is superior to that of IPv6. This is most obvious in the experiments that involve mobility, as the soft handoff mechanism allows the mobile node to maintain the previous connectivity until all of the correspondent nodes have acknowledged the change in connectivity. However, this effect is also present to a lesser degree in the temperature sensor networks which do not involve mobility. This is due to the fact that ILNP's propensity for multipath communication allows for a better distribution of the load across the network and reduces the likelihood of a single node becoming congested and taking a long time to process a packet, or even worse, drop it entirely.

The temperature sensor networks experience less reading delay when using ILNP due to its propensity for multipath too. However the animal tracker and emergency communication network do not exhibit any differences in their application layer delay measurements, as the delay benefits of multipath communication experienced by ILNP are cancelled out by the transmission of locator updates and acknowledgements which takes time to process.

All applications except the emergency communication network experience lower packet and byte overhead when using ILNP due to the superior application layer delivery rate. The reason the emergency communication network experiences higher packet and byte overhead when using ILNP is because every node is mobile, so every node sends locator updates. It is worth noting the locator updates are necessary for soft handoff, and therefore are responsible for the superior application layer delivery rates.

The CPU usage recorded in the nodecentric TSN is higher for IPv6, yet the datacentric TSN records higher CPU usage when ILNP is used, despite the applications and network protocol codebases being virtually identical. Additionally, the graphs showing physical memory usage in single runs show arbitrary increases and decreases which do not appear to correspond to any memory operation in the application or emulated network protocol. This suggests the resource usage data obtained in the experimental scenario applications is very heavily influenced by the effects of other processes and architectural coincidence. Therefore the data cannot be used to conclude which protocol results in lower resource consumption. However given that the codebases of both the emulated network protocols are very similar, and the results for CPU usage and RSS are quite similar over 15 runs for each experiment, the strongest conclusion that can be made is that these emulated protocols result in roughly equivalent resource consumption.

Therefore, since IPv6 can be run on resource-constrained IoT devices, this means that resource-constrained IoT devices could also run ILNP. Therefore, it can be concluded that ILNP-enabled IoT devices are more suitable than IPv6-enabled IoT devices for implementing ubiquitous computing, as overall they perform better while running applications which use increasing amounts of ubiquitous communication in the form of multipath communication and mobility.

12.2 Summary

In this project, an emulation of ILNP-enabled IoT devices using an overlay network developed in Python was implemented, and an API which could be used to build applications on this overlay network, perform control and management functions, and return resource usage statistics was developed too. Finally, the code was licensed and packaged with a User Manual and a README so third parties can download and use it.

The implementation of the overlay network and API was fairly complex however through careful planning and design, a well-functioning final product was produced.

A series of experimental scenario applications were developed using this API, and they work as expected. These applications demonstrate:

- Management of more than ten nodes
- Creation of initial network topologies
- Nodes joining, leaving and replacing locator sets
- Self-configuration of connectivity information
- Sending and flooding of application layer data

and more.

The results of the emulation over these applications were compared to that of a similar IPv6 emulation in order to assess how suitable these network layer protocols running over IoT devices would be for implementing ubiquitous communication. In generating this data, an easily usable data pipeline was developed which could be used for future users of this project.

12.3 Objectives Completed

Objective P1 was fully satisfied with the implementation of an ILNP emulation which runs as an overlay network. Each node communicates using a network

layer protocol that emulates ILNP.

Objective P2 was fully satisfied by the implementation of an API that allows the user to build applications which a network of emulated nodes can run.

Objective P3 was fully satisfied by implementing control features such as joining and leaving locators in API functions. Additionally, management of the overlay network was implemented with a management server, and API calls which communicate with this server to start and stop emulated nodes, and easily create network topologies by passing a YAML file which defines the topology.

S1 was fully implemented by packing this project with a User Manual, README and open-source licensing.

Finally S2 was fully implemented with API calls which can be used to obtain CPU and memory statistics for a process running an emulated node.

The original objectives of this project were decomposed into a comprehensive requirements specification, and this project comprehensively satisfied all of these requirements as demonstrated in the "Testing" section where all of the requirements which were suitable for testing were shown to be met with tests that passed, and the "Fulfilment of Requirements" section which stated how this project comprehensively satisfied all the rest of the requirements.

Therefore this project comprehensively met all the objectives which were set out at the beginning.

12.4 Critical Appraisal

The "Simplifying Assumptions" subsection details how this emulation of ILNP differs from its specification in the various RFCs published on the topic. This project could be improved by removing these assumptions and implementing a more comprehensive and accurate emulation of ILNP.

Additionally, the codebase of the emulation was developed very quickly and while it performs the expected functionality, there are areas it could be improved on. These are most notably efficiency and error handling, and improving these aspects of the codebase would make the emulation more performant and allow the user to quickly diagnose any problems which arise.

While a management interface was implemented, and the recommended method of running it as a systemd service is very simple and only requires that the submitted unit file be run, this was not able to be demonstrated under the permissions restrictions that the schools labs impose, and instead some scripts which run the management server in a TMUX session had to be used. It would

have been better to be able to show that the management server can be run as a systemd service on a computer on which root access is available.

The API provides an interface for sending and receiving application datagrams, however it would have been good if this could be extended to provide support for sending and receiving application streams.

12.5 Future Work

This project could be improved with access to a dedicated set of computers on which the experimental scenario applications could be run. This would eliminate the effects of other unwanted processes in the experimental results, which particularly affected the resource consumption measurements. Getting more reliable results for how the usage of ILNP and IPv6 affects resource consumption would allow conclusions to be drawn about how viable each protocol is for usage on resource-constrained IoT devices.

Additionally, given access to a dedicated set of computers, delay and packet loss could be emulated using netem to make the link layer characteristics more similar to a wireless environment, therefore the emulation's performance in the scenario applications would more closely mirror the performance of IoT devices in a wireless environent which is how they would be deployed in an ubiquitous computing setting. The reason netem was not used in this project is because it was not possible on the school lab machines as this would have been inconsiderate to other users.

Finally, the emulation does not record energy consumption data as the Python libraries which were investigated to enable this record energy usage of the whole machine, which would not result in data representative of IoT energy consumption as the emulation will normally operate on a desktop that uses a lot of power. The paper called "Modeling and Comparison of Delay and Energy Cost of IoT Data Transfers"[33] presents interesting data about the energy consumed by IoT devices using different technologies, sending different amounts of data over different distances so these values could be used to estimate the energy that the IoT devices being emulated would consume.

References

[1] URL: https://svn.python.org/projects/python/trunk/Demo/sockets/mcast.py.

[2] Argparse - parser for command-line options, arguments and sub-commands. URL: https://docs.python.org/3/library/argparse.html.

[3] Bash script to start a detached tmux session, run a script, and keep the tmux session alive. URL: https://unix.stackexchange.com/questions/586407/bash-script-to-start-a-detached-tmux-session-run-a-script-and-keep-the-tmux-se.

[4] Coroutines and tasks. URL: https://docs.python.org/3/library/asyncio-task.html.

[5] Enum - support for enumerations. URL: https://docs.python.org/3/library/enum.html.

[6] Ipv6 man page. URL: https://man7.org/linux/man-pages/man7/ipv6.7.html.

[7] Psutil documentation¶. URL: https://psutil.readthedocs.io/en/latest/.

[8] Python - multithreaded programming. URL: https://www.tutorialspoint.com/python/python_multithreading.htm.

[9] Python simple socket client/server using asyncio. URL: https://stackoverflow.com/questions/48506460/python-simple-socket-client-server-using-asyncio.

[10] Socket - low-level networking interface. URL: https://docs.python.org/3/library/socket.html.

[11] Streams. URL: https://docs.python.org/3/library/asyncio-stream.html.

[12] Transports and protocols. URL: https://docs.python.org/3/library/asyncio-protocol.html.

[13] Yaml/pyyaml: Canonical source repository for pyyaml. URL: https://github.com/yaml/pyyaml/.

[14] How can you set class attributes from variable arguments (kwargs) in python, Sep 1958. URL: https://stackoverflow.com/questions/8187082/how-can-you-set-class-attributes-from-variable-arguments-kwargs-in-python.

[15] Extracting mac address using python, Dec 2022. URL: https://www.geeksforgeeks.org/extracting-mac-address-using-python/.

[16] Jim Anderson. Speed up your python program with concurrency, Apr 2022. URL: https://realpython.com/python-concurrency/.

[17] Ran Atkinson and SN Bhatti. Rfc 6743: Icmp locator update message for the identifier-locator network protocol for ipv6 (ilnpv6), Nov 2012. URL: https://datatracker.ietf.org/doc/rfc6743/.

[18] Ran Atkinson and SN Bhatti. Rfc ft-irtf-rrg-ilnp-noncev6: Ipv6 nonce destination option for the identifier-locator network protocol for ipv6 (ilnpv6), Nov 2012. URL: https://datatracker.ietf.org/doc/html/rfc6744.

[19] Ran Atkinson, SN Bhatti, and Scott Rose. Rfc 6742: Dns resource records for the identifier-locator network protocol (ilnp), Nov 2012. URL: https://datatracker.ietf.org/doc/rfc6742/.

[20] Randall Atkinson, Saleem Bhatti, and Stephen Hailes. Evolving the internet architecture through naming. *IEEE Journal on Selected Areas in Communications*, 28(8):1319–1325, 2010. doi:10.1109/JSAC.2010.101009.

[21] RJ Atkinson and SN Bhatti. Identifier-locator network protocol (ilnp) architectural description, Nov 2012. URL: https://www.rfc-editor.org/rfc/rfc6740.

[22] RJ Atkinson and SN Bhatti. Identifier-locator network protocol (ilnp) engineering considerations, Nov 2012. URL: https://www.rfc-editor.org/rfc/rfc6741.

[23] Saleem Bhatti. Cs4105 traffic control in routers. URL: https://studres.cs.st-andrews.ac.uk/CS4105/Lectures/wk03/cs4105-network_traffic.pdf.

[24] Saleem Bhatti. Identifier-locator network protocol, Apr 2022. URL: https://ilnp.cs.st-andrews.ac.uk/.

[25] Jason Brownlee. Asyncio gather() cancel all tasks if one task fails, Nov 2022. URL: https://superfastpython.com/asyncio-gather-cancel-all-if-one-fails/.

[26] Jason Brownlee. Asyncio wait_for() to wait with a timeout, Nov 2022. URL: https://superfastpython.com/asyncio-wait_for/#What_is_Asyncio_wait_for.

[27] Jason Brownlee. Python asyncio: The complete guide, Dec 2022. URL: https://superfastpython.com/python-asyncio/.

[28] Justin Ellingwood. How to use systemctl to manage systemd services and units, Jan 2022. URL: https://www.digitalocean.com/community/tutorials/how-to-use-systemctl-to-manage-systemd-services-and-units.

[29] Xiaoxu Gao. From novice to expert: How to write a configuration file in python, Dec 2020. URL: https://towardsdatascience.com/from-novice-to-expert-how-to-write-a-configuration-file-in-python-273e171a8eb3.

[30] Giampaolo. Giampaolo/psutil: Cross-platform lib for process and system monitoring in python. URL: https://github.com/giampaolo/psutil/.

[31] Allan Jay. Number of internet of things (iot) connected devices worldwide 2022/2023: Breakdowns, growth amp; predictions, Jan 2023. URL: https://financesonline.com/number-of-internet-of-things-connected-devices/.

[32] Michael Krasnov. Services in systemd: In-depth tutorial, Jun 2020. URL: https://everyday.codes/linux/services-in-systemd-in-depth-tutorial/.

[33] Silvia Krug and Mattias O'Nils. Modeling and comparison of delay and energy cost of iot data transfers. *IEEE Access*, PP:1–1, 04 2019. doi:10.1109/ACCESS.2019.2913703.

[34] Ramya Mohanakrishnan. Top 10 applications of iot in 2022, May 2022. URL: https://www.spiceworks.com/tech/iot/articles/top-applications-internet-of-things.

[35] Dragoș Niculescu. Communication paradigms for sensor networks. *Communications Magazine, IEEE*, 43:116 – 122, 04 2005. doi:10.1109/MCOM.2005.1404605.

[36] Ditchaphong Phoomikiattisak and Saleem Bhatti. Ip-layer soft handoff implementation in ilnp. *MobiArch 2014 - Proceedings of the 9th ACM MobiCom Workshop on Mobility in the Evolving Internet Architecture*, 09 2014. doi:10.1145/2645892.2645895.

[37] Ditchaphong Phoomikiattisak and Saleem N. Bhatti. End-to-end mobility for the internet using ilnp. *Wireless Communications and Mobile Computing*, 2019:7464179, Apr 2019. doi:10.1155/2019/7464179.

[38] Chao Ren. How to compare/sort python objects?, Sep 2018. URL: https://medium.com/@chaoren/comparing-class-instances-in-python-is-easy-80f487223228.

[39] Lynn Root. Exception handling in asyncio, Jul 2019. URL: https://www.roguelynn.com/words/asyncio-exception-handling/.

[40] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, Jul 1993. doi:10.1145/159544.159617.

[41] Ryo Yanagida and Saleem 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*, UbiComp/ISWC '19 Adjunct, page 1022–1033, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341162.3349315.

13 Appendices

13.1 User Manual

Installation

To install this emulation, simply download and unzip the submitted .zip file onto any machine that will host an emulated IoT device, or make any API calls to these emulated IoT devices.

Dependencies

To run the emulation management server, an emulated IoT device or the provided API, Python 3.9.14 or later is required. This is the version used in the school labs.

Additionally, to create a network of emulated IoTs, the emulated IoTs must run on a set of Linux machines that are IPv6-enabled and on the same IPv6 network. The application using the provided API must be able to communicate with the machines that each emulated IoT operates on over IPv4.

The required python modules can be installed using the provided requirements.txt file by running: `pip3 install -r requirements`.

Running the Emulation Management Server

Once these dependencies have been met, the emulation management servers can be set up in two different ways.

The first option, which is recommended, is to run it as a systemd service that is started when the machine boots up. This is achieved by navigating to the root directory of the project and running the following command[28]: `sudo systemctl enable emulated_iot.service`. This must be done for every machine which is going to run the emulation management server, however once this is done once it should not have to be done ever again.

The second option, which is not recommended, is to run the emulation management servers using TMUX. This was necessary when running the emulation on the school labs, as students are not allowed to run systemd services. Note that TMUX is a dependency for this method. This is achieved with the following steps:

- Set up an SSH key on all of the machines on which the emulation management server is going to be run

- Restart the TMUX sessions hosting the emulation management servers by navigating to the root of the project directory and running `refresh_TMUX_servers.py <hostnames_file> <port> <project_root_dir>` where:
  - `hostnames_file` is a file that contains the hostnames of the hosts which will run the emulation management servers on each line
  - `port` is the port that the emulation management server will run on
  - `project_root_dir` is the absolute pathname to the root project directory. This must be valid for the machine on which this script is being run and all of the hosts on which the emulation management server is being started.

  This command can be re-run to ensure that all the specified hosts are running the servers as many times as the user likes.

Now each host which should be running the emulation management server will be running it in a TMUX session. If a host in the hosts file is unreachable, a timeout is configured so that it does not stop the script from starting the emulation management server on other hosts.

Configuration File

The emulated IoTs take runtime parameters from the YAML file called "config.yml", located in the src/ directory. The meanings of the parameters are as follows:

Field	Meaning
`mcast_prefix`	The routing prefix which is combined with the emulated locator values to form the underlying IPv6 multicast addresses. Format: xxxx:xxxx:xxxx:xxxx
`mgmt_port`	The port that the emulated IoT's API server operates on.
`emulation_port`	The port that the emulated IoTs communicate with each other over IPv6.
`router_ad_interval`	Number of seconds between consecutive router advertisements being broadcast.
`neighbour_ad_interval`	Number of seconds between consecutive neighbour advertisements being flooded.
`lu_resend_interval`	Number of seconds between consecutive locator update retransmissions.
`purge_rate_limit`	The minimum number of seconds between consecutive runs of a method that purges soft state.
`own_loc_active_timeout`	The number of seconds for which an emulated IoT's locator can be `ACTIVE` after its most recent advertisement by a router advertisement.
`own_loc_aged_timeout`	The number of seconds for which an emulated IoT's locator can be `AGED` before being purged.

correspondents_timeout	The number of seconds for which an emulated IoT records a different emulated IoT as a correspondent after their most recent unicast communication.
forwarding_entry_timeout	The number of seconds a forwarding entry is stored before it is purged.
remote_loc_active_timeout	The number of seconds a remote I-LV mapping is kept ACTIVE.
remote_loc_aged_timeout	The number of seconds a remote I-LV mapping is kept AGED before it is purged.
remote_loc_expired_timeout	The number of seconds a remote I-LV mapping is kept EXPIRED before it is purged.
available_router_timeout	The number of seconds a router is stored as a valid router after its most recently received router advertisement.
BROADCAST_LOC	Locator on which all emulated IoTs are reachable. Format: 0x{hex_value}
BROADCAST_IDENTIFIER	The identifier which represents all nodes on a given locator. Format: 0x{hex_value}
QUANTUM	Quantum for the deficit round-robin multipath algorithm.
FLOW_INFO	Flow identifier of the underlying IPv6 multicast addresses. [10][6]
SCOPE_ID	The identifier of the scope of the underlying IPv6 multicast addresses used.[10][6]
last_used_timeout	Number of seconds that needs to pass since its last usage for a locator to no longer be considered recently used, which determines the difference between EXPIRED and AGED
startup_wait	Number of seconds the emulated IoT waits after sending neighbour and router solicitations during startup.
delay	The number of seconds between the emulated IoT process receiving and processing a datagram.
max_locs	The maximum number of locators which can be used in an advertisement or locator update.
transport_mss	Maximum number of application bytes that can be sent in a single transport datagram. Fragmentation occurs automatically.

When installing and running on a new network, ensure that the mcast_prefix refers to an accessible multicast prefix.

To choose the interface which the emulated IoT nodes should use, set the SCOPE_ID configuration parameter to use the parameter with the passed scope ID. The scope ID of each interface on a machine can be viewed using the ip addr command.

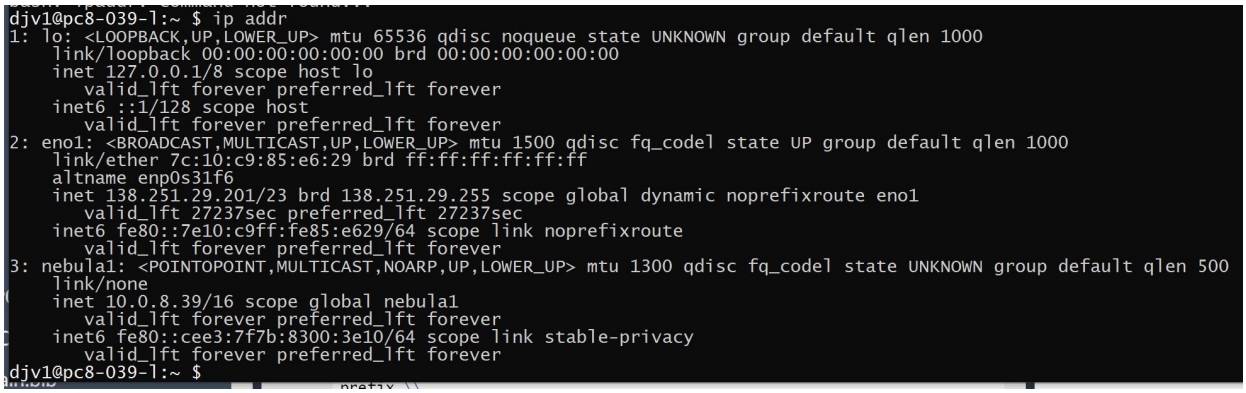

Figure 67: Diagram showing the output of the ip addr command and how it shows the scope id of each interface

Figure 67 shows that the eno1 interface has the scope ID 2.

13.1.1 API

The API commands are contained in the emulation_api.py file and they can be used in a Python script that the user wrote by adding the import statement: `from emulation_api import *` (ensuring that src/ is in the path, if it is not it can be added to the path using sys.path.append).

The documentation for the API calls can be found in the "Emulation API Pydoc" appendix.

13.1.2 Running experiments

Each experiment can be run by navigating to that experiment's folder (which will be one of the subfolders in the src/experiments folder) and running the `run_experiment.py` script. The arguments this script takes and the application layer configuration can be found in that experimental scenario application's subsection in the "Experimental Scenario Applications" section of this report.

13.2 Emulation API Pydoc

emulation_api

index
/cs/home/djv1/Documents/CS4099/ilnp_iot-djv1/src/emulation_api.py

Modules

asyncio	pickle	sys	uuid
math	socket	time	yaml
os	struct	traceback	

Classes

builtins.object

- IoTDevice
- IotSocket

class **IoTDevice**(builtins.object)

```
IoTDevice(hostname, port, net_protocol)

Class representing emulated IoT device
```

Methods defined here:

__init__(self, hostname, port, net_protocol)
```
Constructor
```

async **create_socket**(self, port, reader, writer)
```
Creates a socket for the emulated IoT on the given port and returns it
```

async **get_available_routers**(self, reader, writer)
```
Get which routers have recently sent a router advertisement to this emulated IoT device
```

async **get_command**(self, item_index, reader, writer)
```
Private method which enables generalised get command
```

async **get_correspondents**(self, reader, writer)
```
Get the emulated IoTs that have recently engaged in unicast communication with this emulated IoT device
```

async **get_cpu_time**(self, reader, writer)
```
Get the CPU time of the process running the emualted IoT device
```

async **get_forwarding_table**(self, reader, writer)
```
Get the current state of this emulated IoT's forwarding table
```

async **get_id**(self, reader, writer)
```
Get the identifer of the emulated IoT device
```

async **get_ilv_cache**(self, reader, writer)
```
Get the current state of this emulated IoT's ILV cache
```

async **get_inv_locs**(self, reader, writer)
```
Get the invariant locators of the given emulated IoT device
```

async **get_iot_device_conn**(self)
```
Gets streams which are connected to the IoTDevice which are required to make API calls over the network
```

async **get_locs**(self, reader, writer)
```
Get all locators of the given emulated IoT device
```

async **get_overhead**(self, reader, writer)
```
Get the overhead statistics of the emulated IoT device
```

async **get_rss**(self, reader, writer)
```
Get the current RSS of the process running the emulated IoT device
```

async **get_runtime**(self, reader, writer)
```
Get the length of time this emulated IoT device has been running
```

async **get_value_command**(self, value_index, reader, writer)
```
Private method which enables generalised get commands
```

async **get_virtual_mem**(self, reader, writer)
```
Get the current size of the virtual memory of the process running the emulated IoT device
```

async **is_router**(self, reader, writer)
```
Returns a boolean representing whether the emulated IoT device performs routing/forwarding
```

async **join**(self, reader, writer, *args)
```
Attempts to add the passed in locators to the emulated IoT's locator set
```

async **leave**(self, reader, writer, *args)
```
Attempts to remove the given locators from the emulated IoT's locator set
```

async **quit**(self, reader, writer)
```
End this emulated IoT device and make the hosting machine return to running the emulation management server
```

async **replace**(self, reader, writer, *args)
`Attempts to replace the emulated IoT's locator set`

Data descriptors defined here:

__dict__
`dictionary for instance variables (if defined)`

__weakref__
`list of weak references to the object (if defined)`

class **IotSocket**(builtins.object)

`IotSocket(iot_device, port)`

`Transport layer socket of an emulated IoT device`

Methods defined here:

__init__(self, iot_device, port)
`Socket constructor`

async **close**(self, reader, writer)
`Closes the socket`

async **flood**(self, dst_port, data, reader, writer)
`Flood the passed data throughout the network, on the passed port`

async **has_data**(self, reader, writer)
`Returns whether the given socket has any more data which can be returned by a call to recv.`

async **recv**(self, reader, writer)
`Returns the oldest packet of data that has been sent to it that has not yet been returned`

async **send**(self, dst_id, dst_port, data, reader, writer)
`Sends the given data to the node with the given identifier, on the given port`

Data descriptors defined here:

__dict__
`dictionary for instance variables (if defined)`

__weakref__
`list of weak references to the object (if defined)`

Functions

async **cancel**(hostname, server_port)
`Cancels the task running an emulated IoT`

async **check_servers**(hostsfilename, port, timeout)
`Returns which hosts are available to host an emulated IoT and which are not, out of the hosts listed in the passed hosts file`

async **close_iots**(iots)
`Closes the iots which were created in the create topology call if a later attempt to start another emulated IoT failed`

async **create_topology**(config_file, net_protocol)
`Creates a network of emulated IoT devices with the topology specified in the topology file referred to in the passed config file`

`The emulated IoTs will run over the passed net_protocol too`

async **get_iot_device**(hostname, server_port, emulation_port, no_inv_locs, net_protocol)
`Sends request to emulation server on (hostname, server_port) to start an instance of the emulated device which uses the specified net_protocol and has the specified number of invariant locators (-1 should be passed if the node should not take part in forwarding)`

`The emulation_port argument specifies the port which the emulated IoT device will service API calls on`

async **get_running_iot**(hosts, settings_dict, no_of_locs, net_protocol)
`Start an emulated IoT device on one of the hosts in the passed hosts file, which runs over the network protocol specified, and has the number of invariant locators passed (-1 should be passed if the node is not going to take part in forwarding).`

`The settings_dict passed will determine the port it will use to check for emulation servers, the handshake timeout and the port that the emulated IoT's will service API requests on`

async **handshake**(hostname, server_port, timeout)
`Handshakes with the emulation management server to determine whether it is available to run an emulated IoT`

Data

HANDLER = 'handler'
HeaderSizes = {'ILNP': 40, 'LINK': 24, 'LOCATOR_UPDATE': 16, 'NEIGHBOUR_AD': 24, 'ROUTER_AD': 16, 'TRANSPORT': 12}
RUNNER = 'runner'
SERVER = 'server'
UDP_MAX_SIZE = 65535
VALUE_SIZES = {0: 8, 1: 1, 2: 8, 3: 8, 4: 8, 5: 32, 6: 8}
cmd_sizes = {0: 1, 1: 1, 2: 24, 3: 8, 4: 8, 5: 8, 6: 8, 7: 8, 8: 1, 9: 16, ...}

required_configs = {'BROADCAST_IDENTIFIER', 'BROADCAST_LOC', 'FLOW_INFO', 'QUANTUM', 'SCOPE_ID', 'available_router_timeout', ...}

type_to_decoder = {0: <function decode_noarg_command>, 1: <function decode_noarg_command>, 2: <function decode_send_command>, 3: <function decode_recv_command>, 4: <function decode_sock_command>, 5: <function decode_sock_command>, 6: <function decode_lu_command>, 7: <function decode_lu_command>, 8: <function decode_noarg_command>, 9: <function decode_flood_command>, ...}

13.3 Updated Semantics for Locator Updates

Workgroup:	Network Working Group
Internet-Draft:	draft-bhatti-ilnp-lu-semantics-00
Updates:	RFC6743, RFC6745 (if approved)
Published:	28 October 2021
Intended Status:	Experimental
Expires:	1 May 2022
Author:	S.N. Bhatti *University of St Andrews*

Semantics for ILNP Locator Update (LU) Messages

Abstract

This document defines updates to the Identifier Locator Network Protocol (ILNP) Locator Update (LU) messages, as defined in [RFC6745] and [RFC6743]. The updates are confined to the addition and definition of new values for the "Code" field in the LU header. These new "Code" values make explicit the intention of the LU sender, and the way in which the Locator values in the LU message should be interpretted at the LU receiver. The new "Code" values, effectively, define a set of primitive operation requests that the LU sender sends to its correspondent node. These primitive operations can be used to realise functionality specific to application domains such as: node and network mobility; node and network multihoming; node and network failover; node adn network load-balancing; and node and network moving target defence.

**** XK This document is not yet complete. ****

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 May 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction
   1.1. Requirements Language
   1.2. Rationale
   1.3. Background information
2. Summary of changes to LU header format
3. New stucture for "Code" field
   3.1. Summary of new values for "LU Action"
   3.2. None: 0 (zero)
   3.3. Unused: 1 (one)
   3.4. Add Locator(s) - hard update
   3.5. Add Locator(s) - soft update
   3.6. Delete Locator(s) - hard update
   3.7. Delete Locator(s) - soft update
   3.8. Replace Locator Set - hard update
   3.9. Replace Locator Set - soft update
4. Locator Update Number ("LU_N") field
   4.1. Values for LU_N field
5. Security considerations
6. Privacy considerations
7. Updates in context
   7.1. Updates to RFC6743
   7.2. Updates to RFC6745
8. IANA Considerations
9. Acknowledgements
10. References
   10.1. Normative References
   10.2. Informative References

Author's Address

1. Introduction

The Identifier Locator Network Protocol (ILNP) is described in [RFC6740] to [RFC6748]. Specifically, [RFC6743] and [RFC6745] define a Locator Update (LU) message for use with ILNPv6 (ILNP realised as a superset of IPv6) and ILNPv4 (ILNP realised as a superset of IPv4), respectively. The LU message format has a "Code" field, which only had a single value of zero, and which had no semantics defined. This document defines new values for the "Code" field along with associated semantics. The value of "zero" for the "Code" field is maintained for backwards compatibility, but its use is clarified.

In both ILNPv6 and ILNPv4, an Identifier is a Node Identifier (NID). In ILNPv6, a 64-bit Locator is used, L64. In ILNPv4, a 32-bit Locator is used, L32. All examples in this document are based on the use of ILNPv6, i.e. all examples use NID and L64 values.

For the sake of completeness, this document refers to both ILNPv6 and ILNPv4. However, at the time of writing, only ILNPv6 is actively being developed and deployed.

1.1. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Rationale

LU messages are signalling messages that enable an ILNP node to notify a correspondent node (CN) of its own Locator values. Changes in Locator values typically occur when the connectivity of a node changes, for example with operations concerning application areas such as mobility, multihoming, failover, and load-balancing, but may happen for other, application-specific, reasons. However, it is not currently possible for ILNP nodes to indicate explicitly whether locator updates are "hard" (occuring immediately at the sender of the LU) or "soft" (requiring some loose synchornisation between sender and receiver of the LU), or to indicate the way in which new Locator values should be treated at the CN.

This document defines new values for the "Code" field of LU messages to allow an ILNP node to more precisely and explicitly signal to a CN its intention for the use of new Locator values that are sent in a LU Advertisement message. The new codes are used to define primitive operations - "add", "delete", "replace" - which can be used as building blocks for more complex semantics related to changes in Locator values. We expect such semantics to be defined in additional documents for specific application domains, e.g. mobility, multihoming, failover, load-balancing, etc.

1.3. Background information

The current definition of ILNP is contained in 9 Experimental RFC documents: [RFC6740] and [RFC6741] contain the overall description of ILNP. [RFC6742] contains information about new DNS resource records. [RFC6743] and [RFC6744] contain extensions to IPv6 to enable ILNPv6. [RFC6745], [RFC6746], and [RFC6747] contain extensions to IPv4 to enable ILNPv4. [RFC6748] contains information about various application scenarios when using ILNP.

2. Summary of changes to LU header format

Both [RFC6743] and [RFC6745] and define a header format for LU messages. That format is modified as shown in Figure 1.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Num of Locs  |   Operation   |          LU Number            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+- -                       Locator [1]                       - -+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Preference [1]         |           Lifetime [1]        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+- -                       Locator [2]                       - -+
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Preference [2]         |           Lifetime [2]        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               .                               |
|                               .                               |
```

Figure 1: Locator Update ICMP header, as modified in this document.

This new header format is the same as previously, apart from the following two changes.

1. New values and semantics for the field named "Code" are defined in this document. The new values are set by the sender of the LU message to indicate exlicitly the semantics of this LU. The value of this field was previously 0 (zero).
2. The field named "LU Number" (LU_N) is defined in this document. LU_N is a Locator Update Number, an unsigned 16-bit value that is generated by the sender of the LU Advertisement, and replicated in the coresponding LU Acknowledgement from the CN. This field was previously marked as RESERVED.

The other fields in the header retain the definitions previously documented in [RFC6743] and [RFC6745].

3. New stucture for "Code" field

The purpose of the new stucture and LU Code values for the "Code" field are to define more precise semantics for LU Advertisements, and, therefore, the intention of the sender.

The structure of the code field is modified as shown in Figure 2.

```
  0       1
  8   9   0   1   2   3   4   5
+---+---+---+---+---+---+---+---+
| S |        LU Action          |
+---+---+---+---+---+---+---+---+
```

Figure 2: Locator Update ICMP header, Code field structure.

The bit marked 'S' is a flag. When set to zero, it indicates that the LU Action **MUST** be applied to all the ILNP sessions at the recieving host for the Source NID of the LU Adevrtisement. When set to one, the LU Action **MUST** be applied only to the single session identified by the tuple <Source NID, Destination NID, Nonce>.

If LU Action has value zero, then S **MUST** also have value zero. If one of the other allowable values of LU Action (see Table 1) are used, S can be set to zero or one as required by the sender of the LU message.

A "Locator Set" at an ILNP node is the set of Locators that are currently in use for communication with a CN.

3.1. Summary of new values for "LU Action"

The use of LU messages for signalling and for handling changes in ILNP connectivity are described in Section 6 of [RFC6741]. The syntax of the LU header and packet format is described in [RFC6743]. This document extends those descriptions.

A summary of the new values for the "LU Action" field is given in Table 1. One of these values **MUST** be used for "LU Action" in the "Code" field of the LU header for ILNPv6 [RFC6743] and ILNPv4 [RFC6745]

"LU Action" value (decimal)	"LU Action" name	Description
0 (zero)	ILNP_LU_NONE	This is the only value defined in [RFC6743] and [RFC6745]. It is kept for backwards compatibility. When LU Action is zero, the S flag **MUST** also be zero. An ILNP node **MUST** not use this value when sending a LU message, but **SHOULD** accept a value of zero when receiving LU messages, for backwards compatibility.
1	ILNP_LU_UNUSED	This value **MUST NOT** be used.
2	ILNP_LU_ADD_HARD	"Add Locator(s) - hard update." Locator value(s) that **MUST** be added to the current Locator set.
3	ILNP_LU_ADD_SOFT	"Add Locator(s) - soft update." Locator value(s) that **MUST** be added to the current Locator set.
4	ILNP_LU_DELETE_HARD	"Delete Locator(s) - hard update." Locator value(s) that **MUST** be deleted from the current Locator Set.
5	ILNP_LU_DELETE_SOFT	"Delete Locator(s) - soft update." Locator value(s) that **MUST** be deleted from the current Locator Set.

"LU Action" value (decimal)	"LU Action" name	Description
6	ILNP_LU_REPLACE_HARD	"Replace Locator Set - hard update." The Locator value(s) that MUST replace the current Locator Set.
7	ILNP_LU_REPLACE_SOFT	"Replace Locator Set - soft update." The Locator value(s) that MUST replace the current Locator Set.

Table 1: A summary of the new values for "LU Action" for ILNP LU messages

The value of the "Code" field in the LU header MUST be one of the values in Table 1, other than ILNP_LU_NONE or ILNP_LU_UNUSED. No other values are permitted for the "Code" field, and are reserved for future use. If any other values are seen at the receiver, then that is an error, the error SHOULD be logged in an appropriate system log, and the LU message SHOULD NOT be processed further. Such reporting would also deter the use of this field as a covert channel.

The manner in which "Add Locator(s)", "Delete Locator(s)", and "Replace Locator Set" are used is determined by the intended application domain, e.g. host mobility, site mobility, host-multihoming, site-multihoming, failover, load-balancing, moving target defence, etc.

The semantics of "hard" and "soft" updates determine the synchronisation between the sender and receiver (CN) of the values in a LU Advertisement. In general, for a "hard" update, the sender starts to use the updated Locator state, and discards the previous Locator state, as soon as it has sent the LU Advertisement. For a "soft" update, the sender starts to use the "new" Locator state as soon as it has sent the LU Advertisement, in parallel with the "old" Locator state, and only discards the "old" Locator state after it has received a LU Acknowledgement from the receiver of the LU. However, the semantics and handling of each of these case is described in its own sub-section below.

3.2. None: 0 (zero)

"Code" value zero, 0, ILNP_LU_NONE

The semantics and handling of this value is undefined, and other methods (such as application-level signalling, application-specific configuration, or out-of-band signalling) would be used to synchronise on handling and semantics of LU messages between communicating ILNP nodes.

This value MUST not be used by a sender, but SHOULD be accepted by a receiver, for backwards compatibility.

3.3. Unused: 1 (one)

"Code" value 1, ILNP_LU_UNUSED.

This value MUST NOT be used.

Note: From the values of the other codes, it is clear that the "soft" update variants all have odd number values, so the least-significant bit of the code value is set when a "soft" update is to be made. This information might be used in an implementation.

It is RECOMMENDED that the reception of this value is reported with appropriate system mechanisms, such as systems logs, and flagged as an error. Such reporting would also deter the use of this bit as a covert channel.

3.4. Add Locator(s) - hard update

"Code" value 2, ILNP_LU_ADD_HARD.

This "Code" value is used in a LU Advertisement when a sender adds one or more Locators to its current active Locator Set for a CN.

The LU Advertisement contains the Locator value(s) that the CN **MUST** add to its current Locator Set for the sender of the LU.

The sender can start to use the new Locator(s) that have been advertised as soon as the LU Advertisement has been sent: the sender does not need to wait for a LU Acknowledgement to start using the advertised Locator(s).

After the CN has updated its Locator Set, it **MUST** start to use the new Locator set at once, and **MUST** send a LU Acknowledgement message which contains the same "Code" value and the same Locator value(s) that were in the corresponding LU Advertisement.

3.5. Add Locator(s) - soft update

"Code" value 3, ILNP_LU_ADD_SOFT.

This "Code" value is used in a LU Advertisement when a sender adds one or more Locators to its current active Locator Set for a CN.

The LU Advertisement contains the Locator value(s) that the CN **MUST** add to its current Locator Set for the sender of the LU.

The sender of the LU Advertisement **MUST** wait to use the new Locator(s) until one of the following conditions have been met:

1. A LU Acknowledgement is received from the CN.
2. A packet is received from the CN using (one of) the new Locator value(s) that was sent in the LU Advertisement. This would indicate that the LU Advertisement was received. The corresponding LU Acknowledgement may have been lost or mis-ordered.

After the CN has updated its Locator Set, it **MUST** start to the use the new Locator Set at once, and **MUST** send a LU Acknowledgement message which contains the same "Code" value and the same Locator value(s) that were in the corresponding LU Advertisement.

3.6. Delete Locator(s) - hard update

"Code" value 4, ILNP_LU_DELETE_HARD.

This "Code" value is used in a LU Advertisement when a sender deletes one or more Locators from its current active Locator Set for a CN.

This LU message contains the Locator value(s) that the CN **MUST** delete from its current Locator Set for the sender of the LU.

The sender of the LU Advertisement can stop using the Locator(s) that have been advertised as soon as the LU Advertisement has been sent: the sender does not need to wait for a LU Acknowledgment to stop using the advertised Locator(s).

An ILNP node MUST send a LU Advertisement when a Locator value is no longer usuable. As Locator values are IPv6 prefixes, if the prefix becomes unusable (e.g. if IPv6 Router Advertisements for the prefix are no longer visible), then the sender may not be able to continue using a particular Locator value, and so MUST inform the CN.

After the CN has updated its Locator Set, it MUST stop using the relevant Locator(s) straight away, and MUST send a LU Acknowledgement message which contains the same "Code" value and the same Locator value(s) that were in the corresponding LU Advertisement.

3.7. Delete Locator(s) - soft update

"Code" value 5, ILNP_LU_DELETE_SOFT.

This "Code" value is used in a LU Advertisement when a sender deletes one or more Locators from its current active Locator Set for a CN.

This LU Advertisement contains the Locator value(s) that the CN MUST delete from its current Locator Set for the sender of the LU.

The sender of the LU Advertisement MUST wait to stop using the advertised Locator(s) until one of the following conditions have been met:

1. A LU Acknowledgement is received from the CN.
2. A Locator value is no longer usuable by thhe sender. As Locator values are IPv6 prefixes, if the prefix becomes unusable (e.g. if IPv6 Router Advertisements for the prefix are no longer seen), then the sender may not be able to continue using a particular Locator value.

After the CN has updated its Locator Set, it MUST stop using the relevant Locator(s) straight away, and MUST send a LU Acknowledgement message which contains the same "Code" value and the same Locator value(s) that were in the corresponding LU Advertisement.

3.8. Replace Locator Set - hard update

"Code" value 6, ILNP_LU_REPLACE_HARD.

This "Code" value is used in a LU Advertisement when a sender wishes to update a CN by sending it a completely "new" Locator Set to use in place of the "old" Locator Set that the CN currently has for the sender.

This LU Advertisement contains the complete set of "new" Locator values that the CN MUST now use as its current Locator Set for the sender of the LU, discarding the whole of the "old" Locator Set that it currently uses for that sender.

The sender of the LU Advertisement can start using the Locator(s) that have been advertised as soon as the LU Advertisement has been sent: the sender does not need to wait for a LU Acknowledgment to start using the advertised Locator(s).

After the CN has updated its Locator Set, it MUST start to the use the new Locator Set at once, and MUST send a LU Acknowledgement message which contains the same "Code" value and the same Locator value(s) that were in the corresponding LU Advertisement.

3.9. Replace Locator Set - soft update

"Code" value 7, ILNP_LU_REPLACE_SOFT.

This "Code" value is used in a LU Advertisement when a sender wishes to update a CN by sending it a completely "new" Locator Set to use in place of the "old" Locator Set that the CN currently has for the sender.

This LU Advertisement contains the complete set of "new" Locator values that the CN **MUST** now use as its current Locator Set for the sender of the LU, discarding the whole of the "old" Locator Set that it currently uses for that sender.

The sender of the LU Advertisement **MUST** wait to discard the "old" Locator Set until one of the following conditions have been met:

1. A LU Acknowledgement is received from the CN.
2. A packet is received from the CN using (one of) the new Locator value(s) that was sent in the LU Advertisement. This would indicate that the LU Advertisement was received. The corresponding LU Acknowledgement may have been lost or mis-ordered.
3. A Locator value is no longer usuable. As Locator values are IPv6 prefixes, if the prefix becomes unusable (e.g. if IPv6 Router Advertisements for the prefix are no longer seen), then the sender may not be able to continue using a particular Locator value.

After the CN has updated its Locator Set, it **MUST** start to the use the new Locator Set at once, and **MUST** send a LU Acknowledgement message which contains the same "Code" value and the same Locator value(s) that were in the corresponding LU Advertisement.

4. Locator Update Number ("LU_N") field

The purpose of this field is to allow the sender of the LU Advertiesment to identify the correct LU Acknowledgement if multiple LU Advertiesments are made using the same <source NID, destination NID> pair. So, the only requirement for this value is that it is unique within the context of any other un-acknowledged LU Adverstisements for a given <source NID, destination NID> pair.

This field contains an unsigned 16-bit value that is assigned by the sender of the LU Advertisement. The CN **MUST** use the same value in the corresponding LU Acknowledgement message.

4.1. Values for LU_N field

This field **MUST NOT** be set to 0 (zero) by sender, but a value of 0 (zero) **SHOULD** be accepted by a receiver for backwards compatibility.

This document does not define a particular method or algorithm for generating values for this field. An implementation is free to choose a suitable method or algorithm, as suits the application domain. For example: the LU_N could be a monotonically increasing value; or a value derived from the number sequence generated by pseudo-random number generator (PRNG).

5. Security considerations

Security considerations remain unchanged, and are as discussed in: Section 9 of [RFC6740]; Section 11 of [RFC6741]; and Section 6 of both [RFC6743] and [RFC6745].

6. Privacy considerations

Privacy considerations remain unchanged, and are as discussed in Section 10 of [RFC6740] and Section 12 of [RFC6741].

7. Updates in context

This section provides a guide as to which specific parts of [RFC6743] and [RFC6745] are to be read in conjunction with the updates provided in this document.

7.1. Updates to RFC6743

Section 2 of [RFC6743] is now updated to include the definition of the "Code" values listed in Section 3, and the definition of the LU_N field as defined in Section 4.

Section 4 of [RFC6743] is now updated to include the descriptions of the use of Locator values in LU messages listed in Section 3.

7.2. Updates to RFC6745

Section 2 of [RFC6745] is now updated to include the definition of the "Code" values listed in Section 3, and the definition of the LU_N field as defined in Section 4.

Section 4 of [RFC6745] is now updated to include the descriptions of the use of Locator values in LU messages listed in Section 3.

8. IANA Considerations

This memo includes a request to IANA to approve the new values of the "Code" values for the Locator Update ICMPv6 [RFC6743] and ICMPv4 [RFC6745] messages, as defined in this document.

9. Acknowledgements

I am grateful for discussions with Ryo Yanagida during which I was persuaded to (finally) (re-)write this document and make it public.

I am grateful for feedback from Francis Dupont on an earlier version of this document.

The formulation of this document has been greatly informed by work undertaken mainly with graduate students at the University of St Andrews - my thanks to them all. The relevant background work is listed below, with all papers linked at the ILNP web site https://ilnp.cs.st-andrews.ac.uk [ILNP_www].

- Mobility: [PURBA2019], [WCMC2019], [AINTEC2016], [WMNC2016], [WiMob2015], [MobiArch2014], [PM2HW2N2013], [MILCOM2012]
- Multihoming: [AINA2014], [MILCOM2014], [MILCOM2012]
- Site / datacentre management: [MILCOM2014], [MILCOM2012]
- Security and privacy: [PURBA2019], [MILCOM2014], [MILCOM2012]
- Transport layer / applications: [PURBA2019], [WCMC2019], [AINTEC2016], [PM2HW2N2013]
- Linux implementation of ILNP: [PURBA2019], [WCMC2019], [netdev0x13], [AINTEC2016], [MobiArch2014]

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6740] Atkinson, RJ. and SN. Bhatti, "Identifier-Locator Network Protocol (ILNP) Architectural Description", RFC 6740, DOI 10.17487/RFC6740, November 2012, <https://www.rfc-editor.org/info/rfc6740>.

[RFC6741] Atkinson, RJ. and SN. Bhatti, "Identifier-Locator Network Protocol (ILNP) Engineering Considerations", RFC 6741, DOI 10.17487/RFC6741, November 2012, <https://www.rfc-editor.org/info/rfc6741>.

[RFC6742] Atkinson, RJ., Bhatti, SN., and S. Rose, "DNS Resource Records for the Identifier-Locator Network Protocol (ILNP)", RFC 6742, DOI 10.17487/RFC6742, November 2012, <https://www.rfc-editor.org/info/rfc6742>.

[RFC6743] Atkinson, RJ. and SN. Bhatti, "ICMP Locator Update Message for the Identifier-Locator Network Protocol for IPv6 (ILNPv6)", RFC 6743, DOI 10.17487/RFC6743, November 2012, <https://www.rfc-editor.org/info/rfc6743>.

[RFC6744] Atkinson, RJ. and SN. Bhatti, "IPv6 Nonce Destination Option for the Identifier-Locator Network Protocol for IPv6 (ILNPv6)", RFC 6744, DOI 10.17487/RFC6744, November 2012, <https://www.rfc-editor.org/info/rfc6744>.

[RFC6745] Atkinson, RJ. and SN. Bhatti, "ICMP Locator Update Message for the Identifier-Locator Network Protocol for IPv4 (ILNPv4)", RFC 6745, DOI 10.17487/RFC6745, November 2012, <https://www.rfc-editor.org/info/rfc6745>.

[RFC6746] Atkinson, RJ. and SN. Bhatti, "IPv4 Options for the Identifier-Locator Network Protocol (ILNP)", RFC 6746, DOI 10.17487/RFC6746, November 2012, <https://www.rfc-editor.org/info/rfc6746>.

[RFC6747] Atkinson, RJ. and SN. Bhatti, "Address Resolution Protocol (ARP) for the Identifier-Locator Network Protocol for IPv4 (ILNPv4)", RFC 6747, DOI 10.17487/RFC6747, November 2012, <https://www.rfc-editor.org/info/rfc6747>.

[RFC6748] Atkinson, RJ. and SN. Bhatti, "Optional Advanced Deployment Scenarios for the Identifier-Locator Network Protocol (ILNP)", RFC 6748, DOI 10.17487/RFC6748, November 2012, <https://www.rfc-editor.org/info/rfc6748>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[AINA2014] Simpson, B. and SN. Bhatti, "An identifier-locator approach to host multihoming", Proc. AINA 2014 - IEEE 28th Intl. Conf. Advanced Information Networking and Applications. Victoria, Canada., DOI 10.1109/AINA.2014.22, May 2014, <https://doi.org/10.1109/AINA.2014.22>.

[AINTEC2016] Phoomikiattisak, D. and SN. Bhatti, "IP without IP addresses", Proc. AINTEC 2016 - 12th Asian Internet Engineering Conf. Bangkok, Thailand., DOI 10.1145/3012695.3012701, December 2016, <https://doi.org/10.1145/3012695.3012701>.

[ILNP_www] Bhatti, SN., "ILNP web site", 2020, <https://ilnp.cs.st-andrews.ac.uk>.

[MILCOM2012] Bhatti, SN. and RJ. Atkinson, "Secure & Agile Wide-area Virtual Machine Mobility", Proc. MILCOM 2012 - 31st IEEE Military Communications Conf. Orlando, FL, USA., DOI 10.1109/MILCOM.2012.6415716, October 2012, <https://doi.org/10.1109/MILCOM.2012.6415716>.

[MILCOM2014] Bhatti, SN., Phoomikiatissak, D., and RJ. Atkinson, "Fast, Secure Failover for IP", Proc. MILCOM 2014 - 33rd IEEE Military Communications Conf. Baltomore, MD, USA., DOI 10.1109/MILCOM.2014.50, October 2014, <https://doi.org/10.1109/MILCOM.2014.50>.

[MobiArch2014] Phoomikiatissak, D. and SN. Bhatti, "IP-Layer Soft Handoff Implementation in ILNP", Proc. MobiArch 2014 - ACM Wkshp. Mobility in the Evolving Internet Architecture. Hawaii, USA., DOI 10.1145/2645892.2645895, September 2014, <https://doi.org/10.1145/2645892.2645895>.

[netdev0x13] Yanagida, R. and SN. Bhatti, "End-to-end networking with ILNP in Linux", Proc. Netdev 0x13, Technical Conf. on Linux Networking, March 2019, <https://netdevconf.org/0x13/session.html?talk-ilnp>.

[PM2HW2N2013] Phoomikiatissak, D. and SN. Bhatti, "Network Layer Soft Handoff for IP Mobility", Proc. PM2HW2N 2013 - 8th ACM Intl. Wkshp. Performance Monitoring, Measurement and Evaluation of Heterogeneous Wireless and Wired Networks. Barcelona, Spain., DOI 10.1109/MILCOM.2012.6415716, November 2013, <https://doi.org/10.1145/2512840.2512843>.

[PURBA2019] Yanagida, R. and SN. Bhatti, "Seamless Internet connectivity for ubiquitous communication", Proc. PURBA 2019, Pervasive Urban Applications Workshop (UBICOMP 2019), DOI 10.1145/3341162.3349315, April 2019, <https://doi.org/10.1145/3341162.3349315>.

[WCMC2019] Phoomikiattisak, D. and SN. Bhatti, "End-To-End Mobility for the Internet Using ILNP", Journal Wireless Communications and Mobile Computing, vol. 2019, no. Article ID 7464179, DOI 10.1155/2019/7464179, April 2019, <https://doi.org/10.1155/2019/7464179>.

[WiMob2015] Phoomikiattisak, D. and SN. Bhatti, "Mobility as a First Class Function", Proc. WiMob 2015 - 11th IEEE Intl. Conf. Wireless and Mobile Computing, Networking and Communications. Abu Dhabi, UAE., DOI 10.1109/WiMOB.2015.7348051, October 2015, <https://doi.org/10.1109/WiMOB.2015.7348051>.

[WMNC2016] Phoomikiattisak, D. and SN. Bhatti, "Control Plane handover Analysis for IP Mobility", Proc. WMNC 2016 - 9th IFIP Wireless and Mobile Networking Conf. Colmar, France., DOI 10.1109/WMNC.2016.7543931, July 2016, <https://doi.org/10.1109/WMNC.2016.7543931>.

Author's Address

Saleem N. Bhatti
University of St Andrews
St Andrews
United Kingdom
Email: saleem@st-andrews.ac.uk

13.4 Preliminary Ethics Self-Assessment Form

Section removed as it contains scanned signatures.

13.5 Description, Objectives and Ethics and Resources Form

Ubiquitous Communication for Internet of Things (IoT)

Description, Objectives, Ethics and Resources (DOER)

Description:

In 1993, Weiser proposed the idea of 'ubiquitous computing' which enhances the user's access to computing power in their surroundings, while the systems responsible are effectively invisible to the user due to their reliability and self-sufficiency [1]. The IP Protocol has limitations which mean it cannot satisfy the networking requirements of 'ubiquitous systems'. For example, the overloading of IP addresses as identifiers and topological locators mean that mobile systems, including many Internet of Things (IoT) devices, are not able to move seamlessly between networks, which is a requirement of 'ubiquitous computing'. A potential solution to this problem is ILNP, which is a protocol with separate Identifier and topological Locator values as opposed to an IP address. The aim of this project is to develop a Python library which uses an overlay network to emulate IoT devices. These emulated IoT devices will use an Identifier-Locator communication protocol over IPv6 to better satisfy the requirements of 'ubiquitous computing'. The user will also be able to control the overlay network through an interface and develop applications for this emulated IoT platform through an API which will be developed over the course of this project.

Primary Objectives:

- Design and implement a library in Python which emulates IoT devices using an overlay network. These emulated IoT devices will use an Identifier-Locator communication protocol over IPv6.
- Design and implement an API to allow users to build applications for this emulated IoT platform.
- Provide an interface for control and management of the overlay network.

Secondary Objectives:

- This library should be packaged and licensed so third parties may use it.
- This library should provide a mechanism to measure resource usage.

Ethics:

No ethical concerns were raised during the completion of the preliminary self-assessment form. The completed form will be uploaded to MMS.

Therefore, there are no ethical concerns with this project.

Resources:

In addition to standard school equipment, and equipment the researcher already has access to, the following special resources will be required to complete this project:

- 3 Raspberry Pis or 3 Gigabyte Brix IoT Ultra Compact PC Kits
- Ethernet Switch

- Shuttle Desktop PC
- Monitor
- Dual NIC

Bibliography:

[1] M. Wesier. "Some computer science issues in ubiquitous computing". Communications of the ACM - Special issue on computer augmented environments: back to the real world CACM, Volume 36 Issue 7, July 1993 Pages 75-84. http://dx.doi.org/10.1145/159544.159617

13.6 Remaining Graphs

13.6.1 Nodecentric TSN

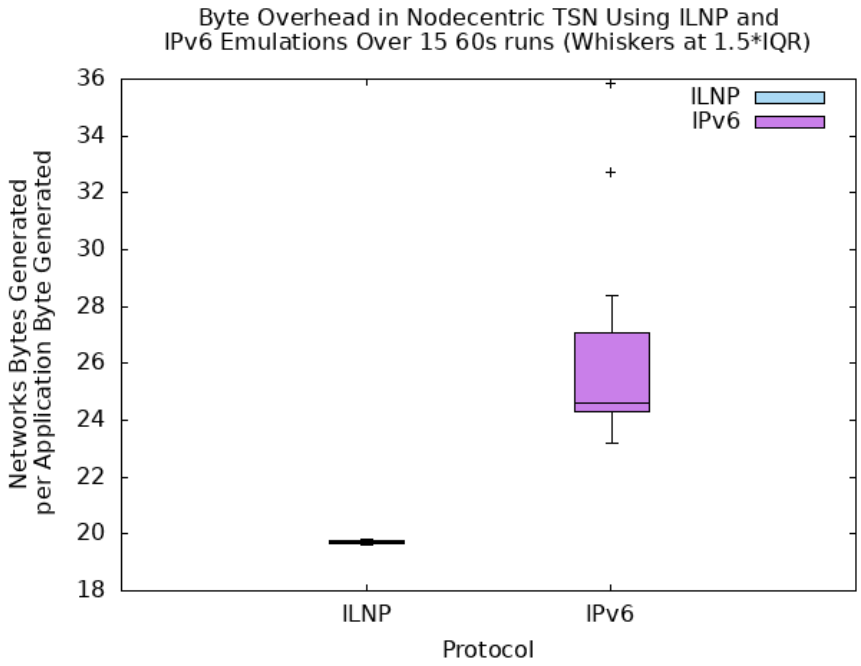

Figure 68: Graph of Byte Overhead in Nodecentric TSN Over 15 Runs

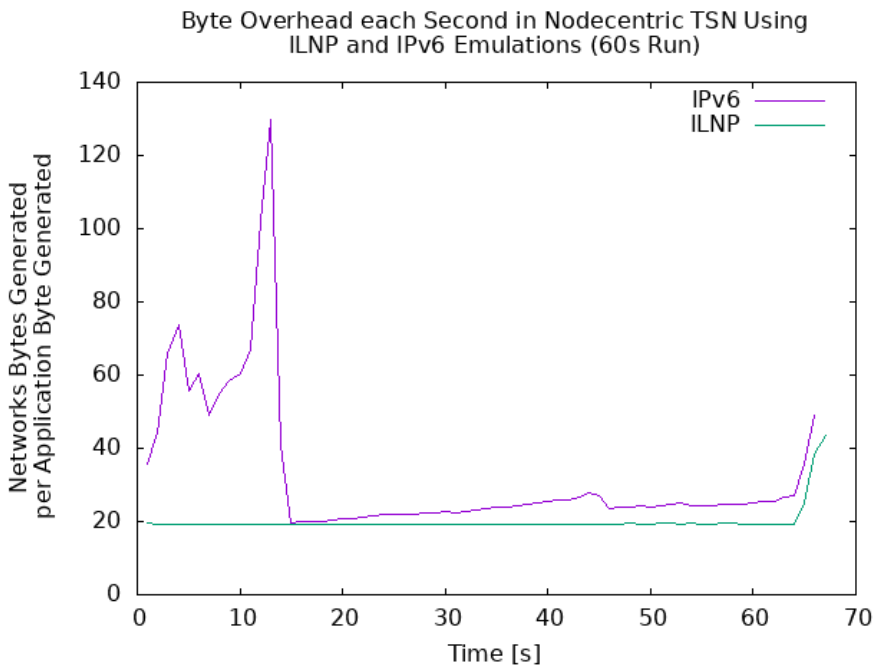

Figure 69: Graph of Byte Overhead per Second in Nodecentric TSN

13.6.2 Datacentric TSN

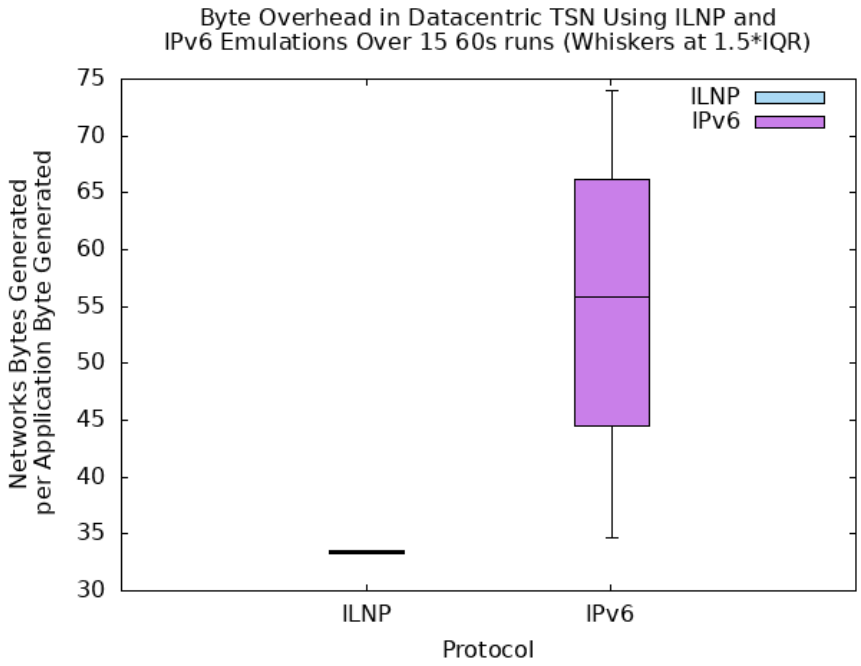

Figure 70: Graph of Byte Overhead in Datacentric TSN Over 15 Runs

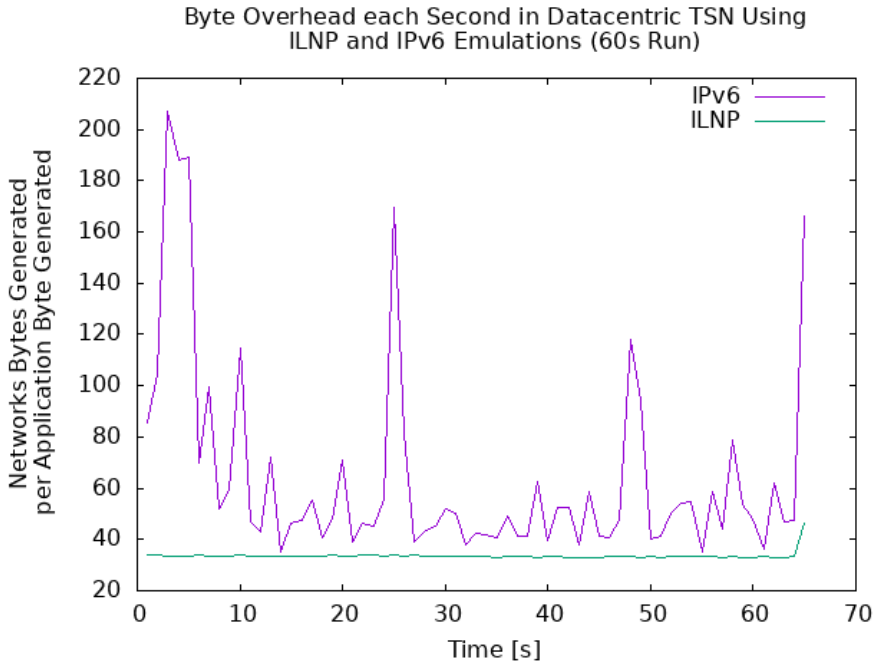

Figure 71: Graph of Byte Overhead per Second in Datacentric TSN

13.6.3 Animal Tracker

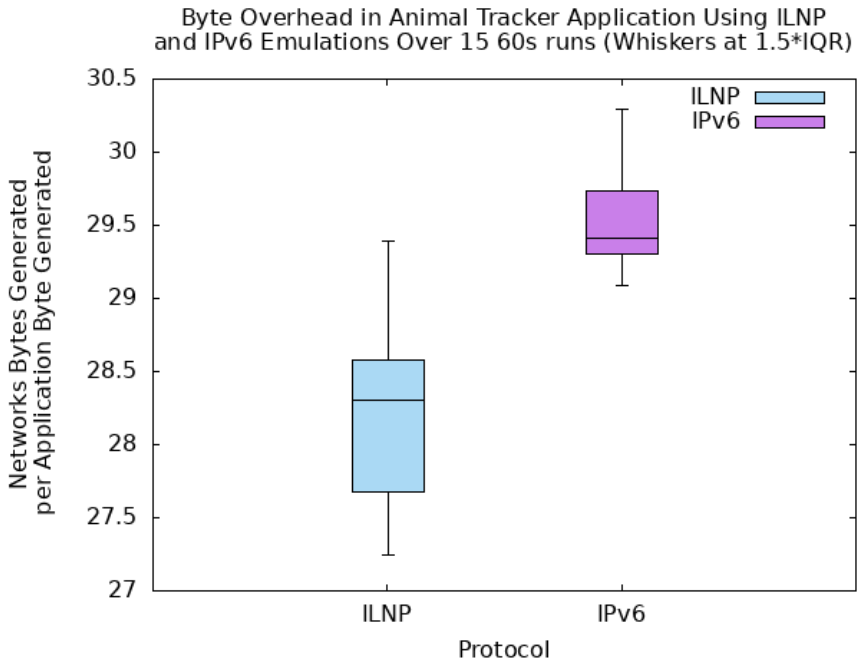

Figure 72: Graph of Byte Overhead in Animal Tracker Over 15 Runs

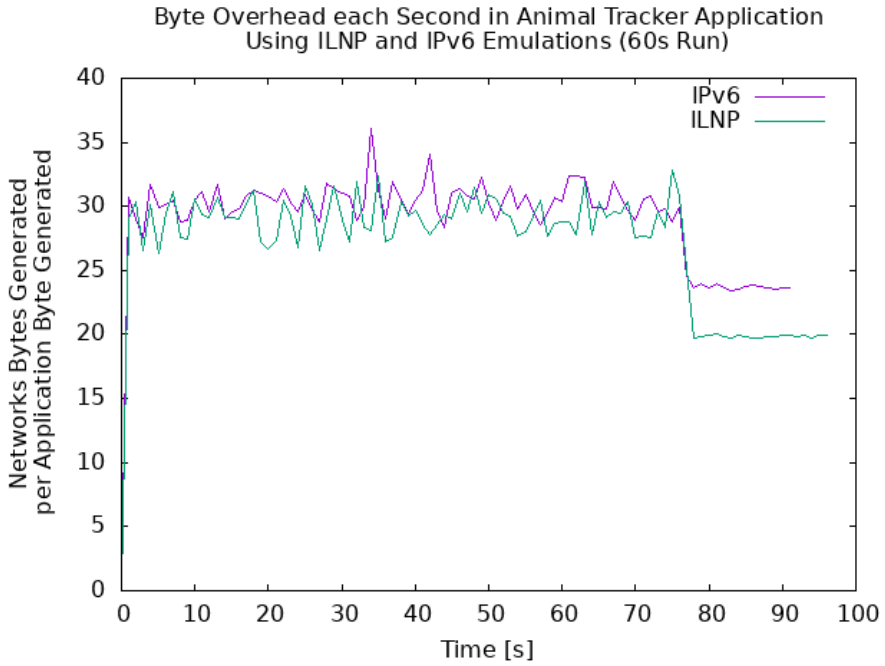

Figure 73: Graph of Byte Overhead per Second in Animal Tracker

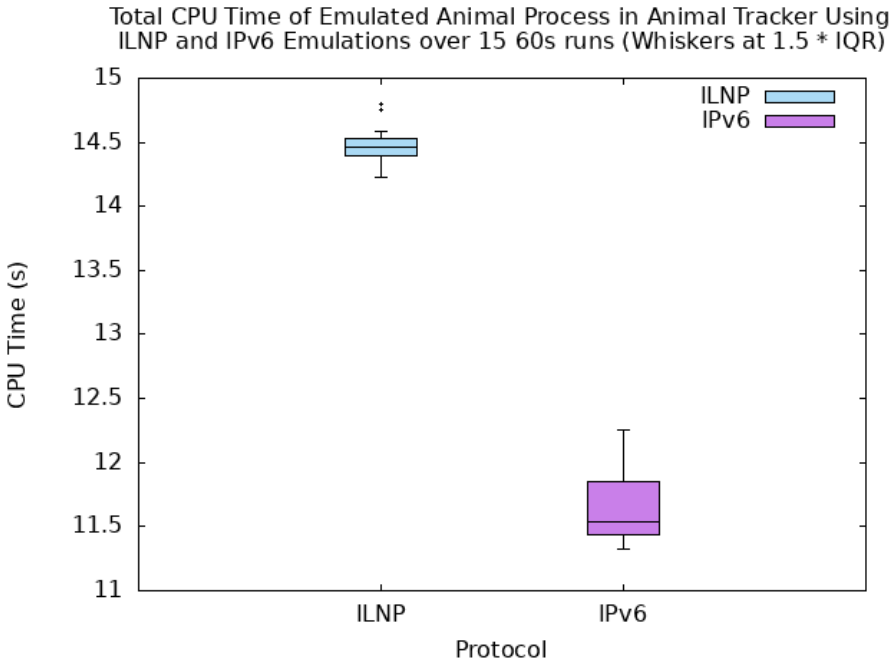

Figure 74: Graph showing animal IoT's CPU usage over 15 runs

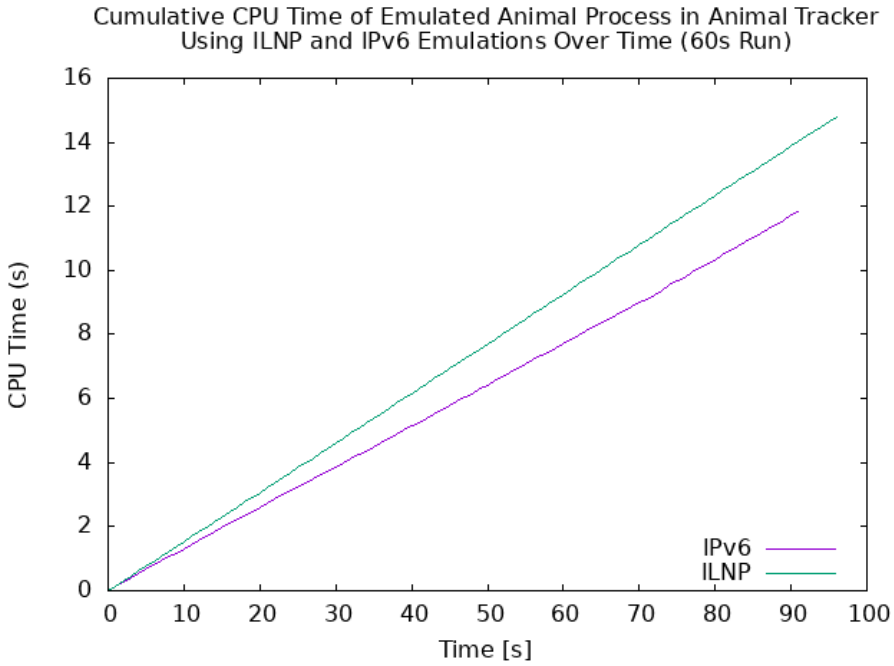

Figure 75: Graph showing animal IoT's CPU usage over a single run

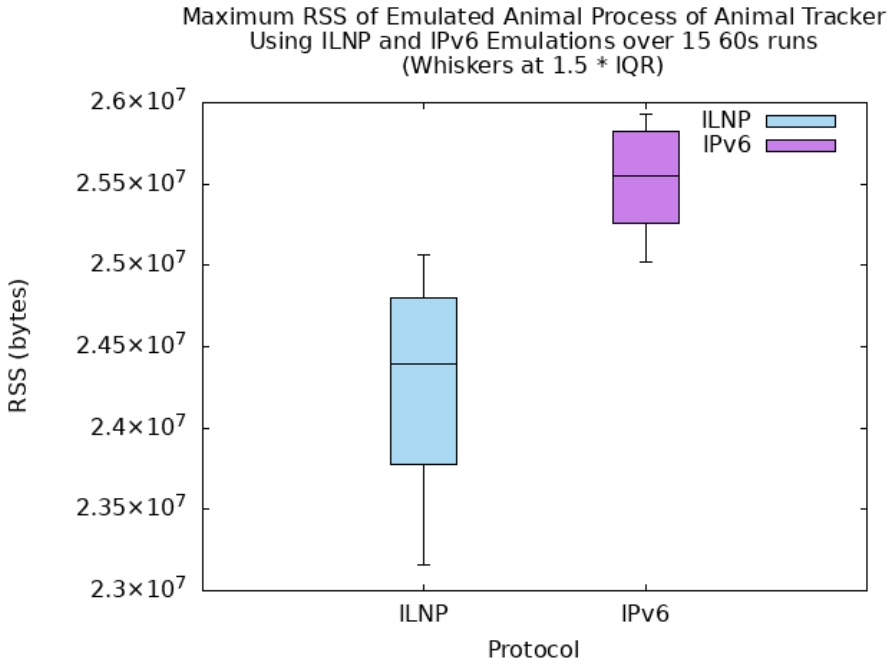

Figure 76: Graph showing animal IoT's RSS over 15 runs

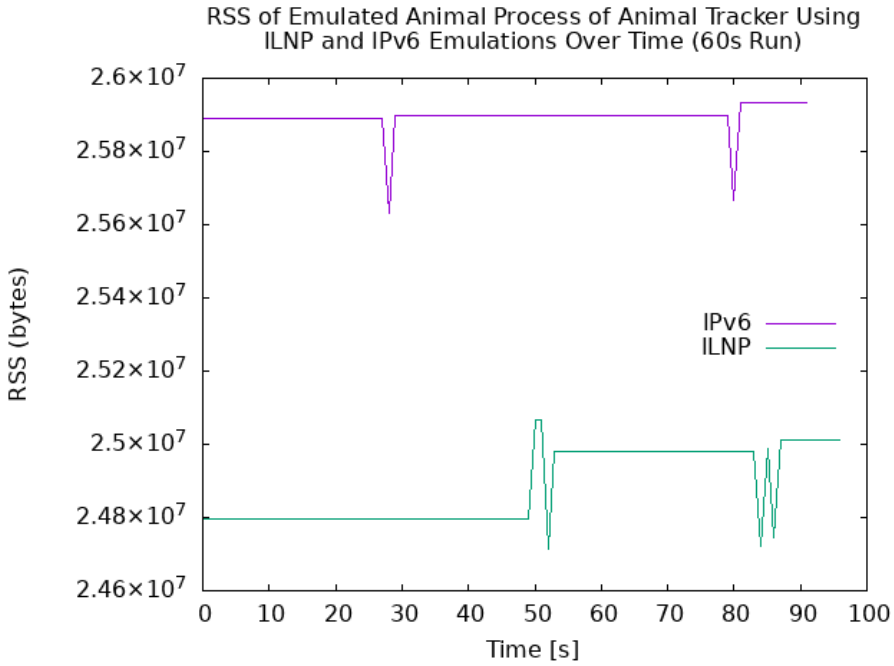

Figure 77: Graph showing animal IoT's RSS over a single run

13.6.4 Emergency Communication Network

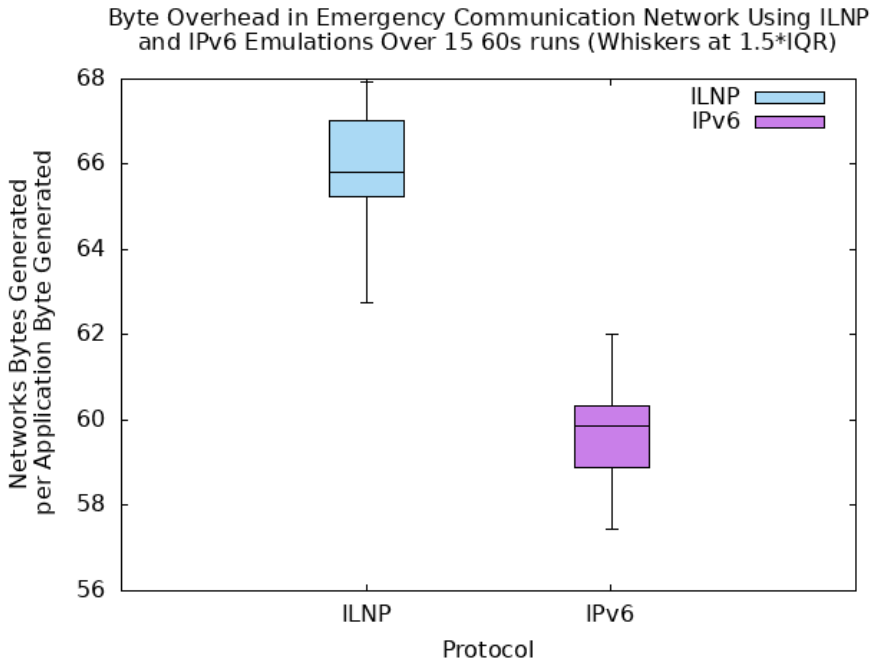

Figure 78: Graph of Byte Overhead in Emergency Communication Network Over 15 Runs

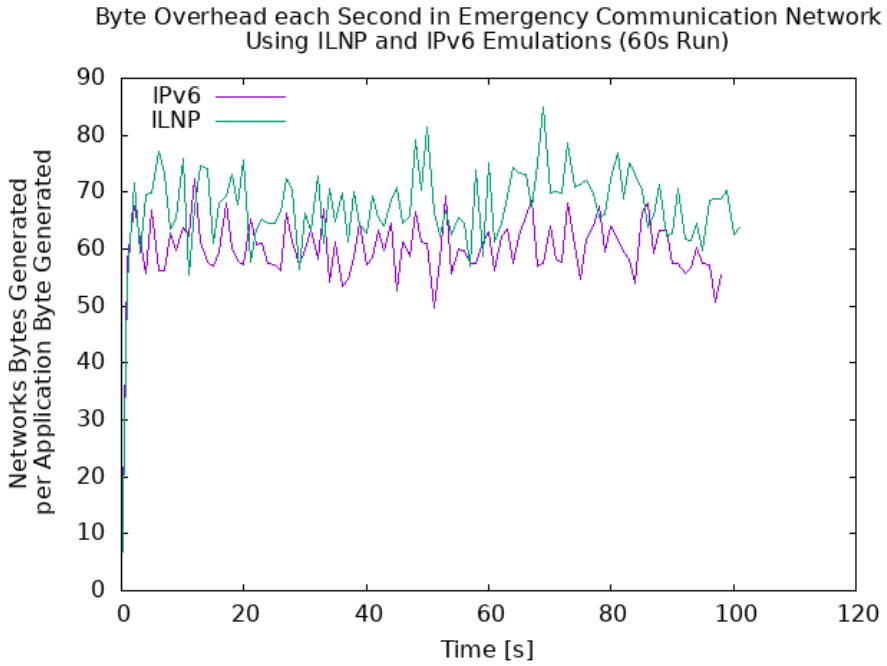

Figure 79: Graph of Byte Overhead in Emergency Communication Network per Second

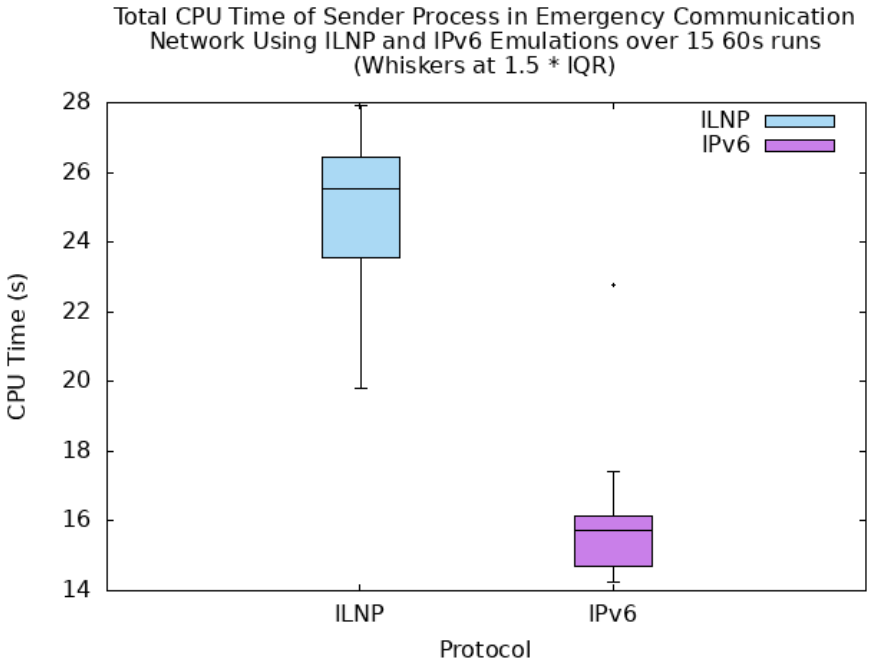

Figure 80: Graph showing sender's CPU usage over 15 runs

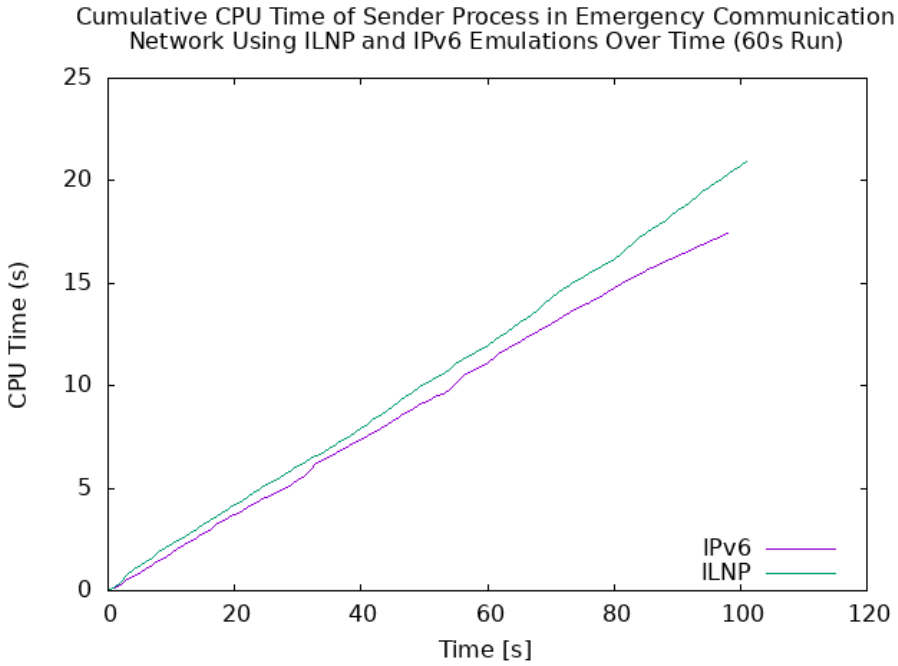

Figure 81: Graph showing sender's CPU usage over a single run

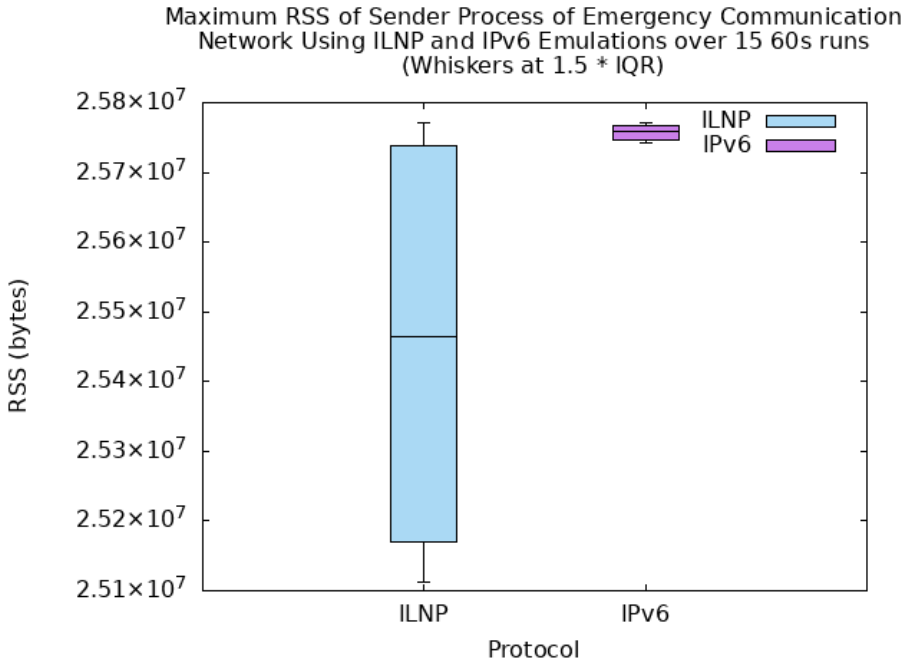

Figure 82: Graph showing sender's RSS over 15 runs

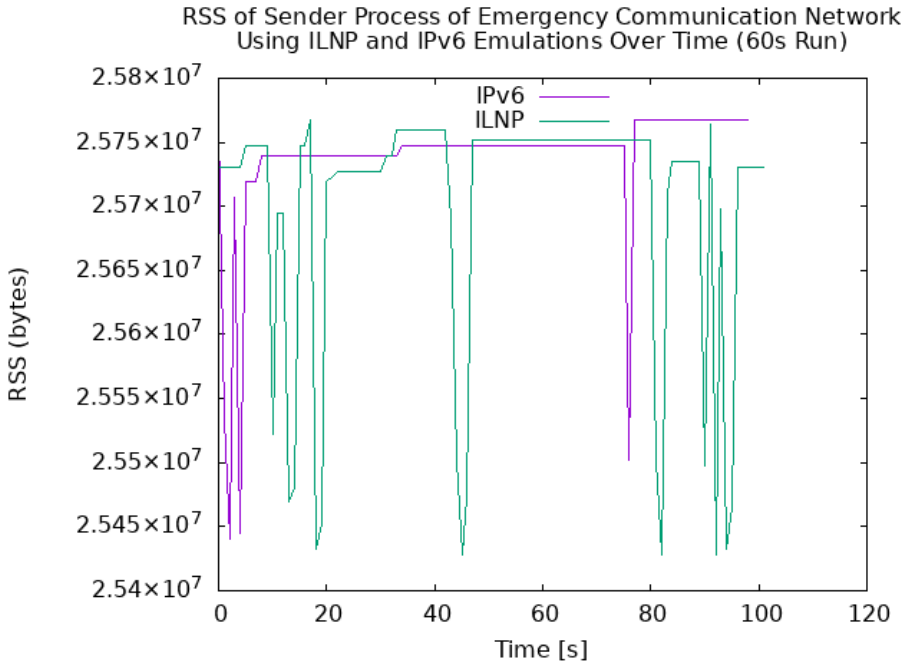

Figure 83: Graph showing sender's RSS over a single run