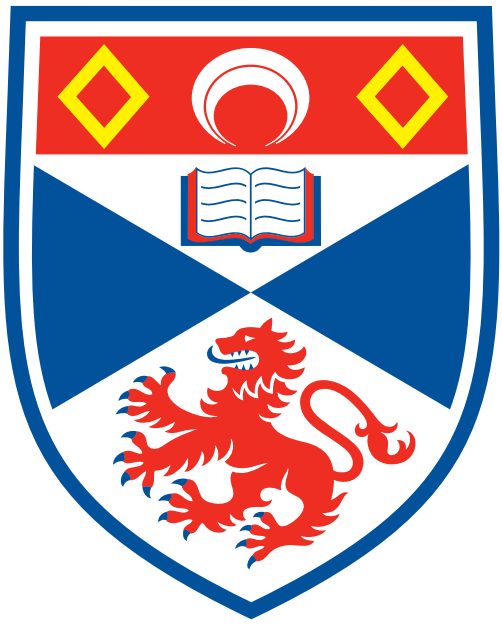
Ubiquitous Communication for the Internet of Things

An Identifier-Locator addressing split overlay network

Author: Ryan Gibb
Supervisor: Saleem Bhatti



School of Computer Science
University of St Andrews
United Kingdom
April 12, 2021

Abstract

The Internet of Things is a modern imagination of Weiser's vision of Ubiquitous Computing. Weiser identified a number of changes to realise this vision, one of which is network support for highly mobile devices. This could not be met by the Internet Protocol 28 years ago, and can not be met by it today. The Identifier-Locator Network Protocol (ILNP) architecture, which has a semantic separation of node's topological location and identify in addressing, has been shown to support highly mobiles by providing seamless connectivity through a layer 3 soft handoff. Experimental analysis of ILNP has only been done on workstations and server machines, however. This works described the design and implementation of an ILNP overlay network built on UDP/IPv6 and its associated protocol operation. An experimental analysis of the operation of the system is done in an IoT scenario with a Raspberry Pi testbed to show how ILNP provides seamless connectivity across network transitions. This demonstrates that the ILNP architecture can successfully provide network mobility support in an IoT context with resource-constrained devices, enabling Weiser's vision of Ubiquitous Computing.

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 11,722 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.

Acknowledgements

I am grateful to my supervisor, Prof. Saleem Bhatti, for his insight and guidance throughout this academic year. His enthusiasm in explaining networking concepts, willingness to devote time to this project every week, and expertise in discussing technical problems, proved invaluable. I am also grateful to the School of Computer Science and the University of St Andrews for an excellent undergraduate education and for providing the opportunity to work on fascinating projects such as this. And finally, I would like to thank my parents for providing continued support and encouragement to pursue higher education.

Contents

1 Introduction

Weiser's 1993 vision of Ubiquitous Computing is a vision of the future of computing where devices are omnipresent and exist in many forms[21]. It has been expanded with the more modern interpretation of the Internet of Things (IoT) which envisions many objects being smart devices interconnected via the Internet. This is essentially the same vision as Weiser's but focused on the physical objects in which the computing is embedded.

Weiser identified a number of changes required to support this vision of ubiquitous computing, some of which were network related. We will refer to these network communication changes required for ubiquitous computing as ubiquitous communication. To enable ubiquitous communication an application should be able to exploit any connectivity available and to move existing communication flows between networks without disruption.[9]

Layer 2 link layer mobility solutions are sufficient for some use cases but rely on being connected to the same network. In order to enable truly ubiquitous communication layer 3 mobility is required, to support devices moving across networks and network types. This is also referred to as a vertical handoff, where that the underlying link layer technology changes. This is opposed to a horizontal handoff where the link layer technology and layer 3 network remain constant, but the network access point changes. For example, one could move between cells in one cellular network with layer 2 mobility, but in order to move to a different network, like another provider's cellular network, or a network using a different technology like IEEE 801.11, layer 3 mobility is required. Indeed this support for different network types is the foundational concept behind the Internet and the Internet Protocol (IP): to enable internetworking, the interconnecting of multiple networks. But IP did not support mobility 28 years ago[21] and doesn't support it to this day.

This mobility should be accessible to applications, which means making the network agility as transparent as possible. Some properties, such as the data rate, won't be able to be maintained across transitions, but the application should not have to deal with changing networks, as is the case with no layer 3 mobility support. Without layer 3 support for mobility, moving between layer 3 networks has to be implemented at the application layer.

A scenario where a solution to this problem would be applicable is a health monitoring system made up of a garment, or several garments, that creates a body area network (BAN). This is an example of a mobile ad hoc network (MANET), but the same principles of mobility still apply. These devices may have to remain connected to each other and a remote server at all times to monitor the health of the individual. The individual will be mobile, and the network connectivity available to the devices will change over time. For example, moving from a cellular network connection to an IEEE 802.11 network when entering an office building where no cellular signal is available. This is especially true in urban environments where there may be a fast-changing large number of network connectivity options available. Additionally, there would be constraints on the resource and energy usage of the devices, and any disruptions to network connectivity would affect this by requiring packet retransmissions.

The largest solution space for this at the moment is implementing mobility through IoT middleware applications, such as TinyLIME/TeenyLIME[11], running on operating systems such as TinyOS[15] and Contiki[13]. Note that solutions to this problem are common to both IoT settings and wireless sensor networks (WSN), so a lot of the literature surrounding them reference the latter.

This kind of software provides a platform for common functionality - including mobility - above the transport layer, with an API for applications to use. It is comparatively very easy to deploy such a middleware solution compared to reworking the network stack. However, the disadvantages of this approach are that it requires the application software to be written for and tied to a specific middleware API, which is rarely standardised. It also adds an additional layer to IoT device's network stacks, which has performance and energy usage implications. If we can provide mobility naively at the network layer it would remove these disadvantages. Additionally, if other functionality of middleware is still required, it would allow for skinnier middleware to be used by implementing part of their functionality - mobility - in the network layer.

Layer 3 mobility can be provided by the Internet Research Task Force architecture called the Identifier-Locator Network Protocol (ILNP)[2][3]. This uses an Identifier-Locator split to allow nodes to move topological location in the Internet whilst maintaining their identifier, and provides a soft handoff for seamless layer 3 network transitions.

This project aims to demonstrate that ILNP can enable ubiquitous communication, focusing on mobility, in the context of IoT devices. This will be done by implementing an ILNP overlay network on top of UDP/IP multicast, and performing experiments based on an evaluative scenario emulating a mobile IoT device. A Raspberry Pi testbed will be used to perform these experiments. Through this we shall show that ILNP can successfully support mobility at the network layer and a seamless transition of communication flows through the use of a soft handoff, thereby enabling ubiquitous communication.

An overlay network was chosen over implementing ILNP in the kernel, as this poses significant technical challenges beyond the time available for this project, and the focus of this project is on the protocol design and interactive over engineering issues. For similar reasons, Python was the language chosen to implement the overlay network in.

1.1 Objectives

Below is listed the original objectives of the project as described in the Description, Objectives, Ethics & Resources (DOER) document.
The primary objectives of the project are:

P1 Produce a python library for an Identifier-Locator communication protocol on an IoT platform. This will take the form of an ILNPv6 overlay network with a skinny transport layer, built on top of UDP Multicast and IP acting as an unreliable link layer.

P2 Create an evaluation scenario based on an emulated IoT application on Linux and IPv6 to demonstrate the operation of the platform.

P3 Describe the security considerations of ILNPv6 for IoT devices; including ephemeral NIDS and locator rewriting relays.

The secondary objectives are:

S1 Measure the multipath effect that can be gained from certain network topologies with ILNPv6's multihoming capabilities contrasting it with existing solutions for mobility.

S2 Create a simple multi-homing policy to best take advantage of the multi-path effect. For example, a scenario where one path to the corresponding node is metered but high bandwidth, while one is not metered but low bandwidth. The paths would be differentiated by locator values.

The tertiary objectives are:

T1 On a Raspberry Pi testbed emulating an IoT platform analyse the performance of the ILNPv6 overlay network contrasting it with existing solutions for mobility.

T2 Compare the energy usage of ILNPv6 and MIPv6 for the IoT. Ideally, this would be real energy measurements from the Raspberry Pi testbed, but could also be estimated from the evaluation scenario statistics.

2 Literature Review

In this chapter, we'll review the literature relevant to this project including mobility in IP, ILNP, the soft handoff, multihoming, security and privacy considerations, and other mobility solutions.

2.1 Mobility in IP

The first non-experimental version of IP, IPv4, was published in September 1981.[19] IBM introduced it's Personal Computer (PC) model 5150 in August 1981, weighting over 9kg.[14] The only way to move these devices was to turn them off, unplug them, transport them, and then set them back up again. The Raspberry Pi model B, used in this project's experiments, weights 45g.[20] Other IoT devices and sensors can weigh much less than this and have inbuilt batteries and wireless connectivity. The Internet was not designed for these kinds of mobile devices. The next major revision of IP, IPv6, did not change the underlying addressing architecture of the Internet.[12]

There are two issues with IP addresses pertinent to mobility. The first is the *overloading* of IP address semantics. IP addresses are used to identify a node's topological location in the Internet through network routing prefixes, as well as to uniquely identify the node in some scope. This becomes an issue when a node changes its location, when it connects to a new network, as it also has to obtain a new IP address.

This wouldn't be an issue in and of itself if a transport (layer 4) flow could dynamically adjust to a new IP address, which brings us to the second issue with IP addresses: the *entanglement* of layers. IP addresses are used both above and below layer 3. Applications use an IP address, usually obtained from a FQDN through DNS, and this IP address is used by the Transport layer, Network layer, and is semi-permanently bound to an interface.

If a node changes location and starts using a new IP address, the application sockets and transport layer flows of any existing connections are broken and have to be re-established. This results in application specific logic being required to deal with layer 3 transitions and the transport layer connection having to be reestablished.

Weiser described this exact problem in 1993:

> "The Internet routing protocol, IP, has been in use for over 10 years. However, neither this protocol nor its OSI equivalent, CLNP, provides sufficient infrastructure for highly mobile devices. Both interpret fields in the network names of devices in order to route packets to the device. For instance, the "13" in IP name 13.2.0.45 is interpreted to mean net 13, and network routers anywhere in the world are expected to know how to get a packet to net 13, and all devices whose name starts with 13 are expected to be on that network. This assumption fails as soon as a user of a net 13 mobile device takes her device on a visit to net 36 (Stanford). Changing the device name dynamically depending on location is no solution: higher-level protocols such as TCP assume that underlying names will not change during the life of a connection, and a

name change must be accompanied by informing the entire network of the change so that existing services can find the device.

A number of solutions have been proposed to this problem, among them Virtual IP from Sony, and Mobile IP from Columbia University. These solutions permit existing IP networks to interoperate transparently with roaming hosts. The key idea of all approaches is to add a second layer of IP address: the "real" address indicating location, to the existing fixed-device address. Special routing nodes that forward packets to the correct real address, and keep track of where this address is, are required for all approaches."[21]

2.2 ILNP

ILNP provides a solution to this problem with an Identifier-Locator addressing split[2]. Instead of addressing nodes with an IP address, we instead use an Identifier-Locator Vector (I-LV). This separates the overloaded semantics of IP addresses into their constituent parts. The identifier uniquely identifies the node, within some scope. The locator provides the topological location of the node in the Internet. Identifiers can be thought of as residing at a locator, and locators thought of as identifying a network.

This solves the first problem of IP address overloading. We can use the identifier and the locator in the transport layer and network layer respectively to solve the issue of layer entanglement as shown in table 2.1. FQDNs are mapped to I-LVs through DNS[7]. Transport layer flow is then dependant on only the identifier, and the network layer is only concerned routing with the locator, which is dynamically bound to an interface.

Protocol layer	ILNP	IP
Application	FQDN	FQDN, IP address
Transport	Identifier, I	IP address
Network	Locator, L	IP address
(interface)	dynamic binding	IP address

Table 2.1: ILNP and IP use of names[8]

Given a TCP connection between two nodes X and Y figure, where P is a port, A is an IP address, L is a locator, and N is an identifier, the protocol state for IP and ILNP is:

$$\text{IP} : \langle tcp : P_X, P_Y, A_X, A_Y \rangle \langle ip : A_X, A_Y \rangle \langle if : A_X \rangle$$
$$\text{ILNP} : \langle tcp : P_X, P_Y, N_X, N_Y \rangle \langle ilnp : (L_X), (L_Y) \rangle \langle if : (L_X) \rangle \quad [9]$$

If node X were to change location and update its locator, the transport and application state would remain unaffected.

ILNP was one of the recommended proposals analysed by the IRTF Routing Research Group:[9]

> "We recommended ILNP because we find it to be a clean solution for the architecture. It separates location from identity in a clear, straightforward way that is consistent with the remainder of the Internet architecture and makes both first-class citizens. Unlike the many map-and-encap proposals, there are no complications due to tunnelling, indirection, or semantics that shift over the lifetime of a packet's delivery."[16]

2.3 ILNPv6

ILNP can be implemented as a superset of IPv6, called ILNPv6.[1][5] The upper 64 bits of the IPv6 address is already used as a routing prefix, which serves the same purpose as an ILNP locator. The identifier is then the lower 64 bits of the IPv6 address. The I-LV corresponds to the IPv6 address as a whole. The syntax is maintained, but the semantics are different. That is, IPv6 addresses and ILNPv6 I-LVs look the same on the wire but are interpreted differently. See figure 2.1.

Well-behaved applications that conform to RFC1958 should not use absolute IP addresses[10]. Therefore modifications to DNS should result in backwards compatibility, as I-LVs are syntactically identical to IPv6 addresses, and the provided mobility is transparent to the transport layer.

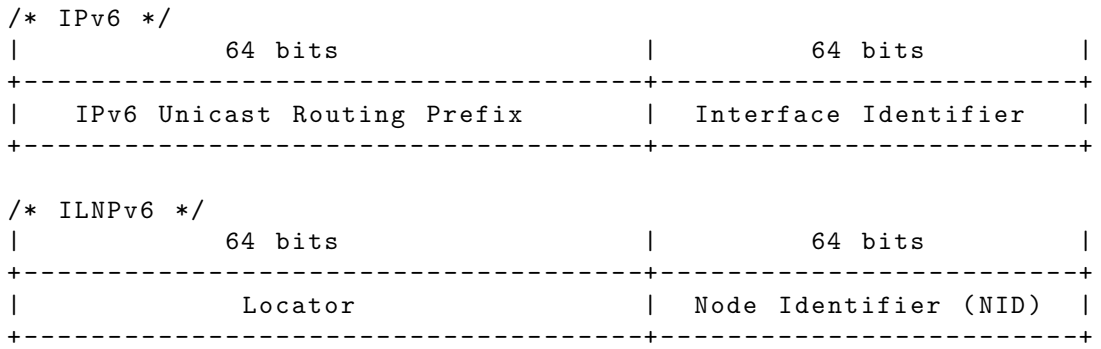


Figure 2.1: IPv6 and ILNPv6 Address Structure[3]

ILNP can also be implemented as an extension to IPv4, ILNPv4[4], but this won't be considered in this report.

2.4 ILNP mobility

Mobility is provided in ILNP through ICMP Locator Update messages[2]. When a node changes network it sends a Locator Update message on its previous network to any hosts in a unicast communication session with it. Then it receives a Locator Update acknowledgement from all such hosts on the new network, verifying that a connection is possible to them over the new network and that they have received the Locator Update message.

ILNP provides seamless connectivity during these network transitions through the use of a soft handoff[18], in the same way a soft handoff is used in code-division multiple access (CDMA) between adjacent cells. After the Locator Update acknowledgement is received, the node retains a connection to the old network for a set duration to receive any packets sent to the node via the old network, and after this then disconnects, completing the handoff process and the network transition. This ensures no loss of packets due to the network transition and enables a seamless transfer of connections.

One might ask what the advantages of this mobility over just re-establishing connections in the application layer are? A seamless layer 3 transition has performance, connectivity, and energy benefits due to the lack of packet loss. The more mobile a device, and the more resource-constrained a device, the more important this becomes. Therefore this can be very important for mobile IoT devices.

Note that this soft handoff requires hardware support in the form of either two network adaptors, or adaptor support for multiple connections on the same adaptor such as through CDMA.

If this is not possible a "break before make" hard handoff is also supported, as opposed to the "make before break" soft handoff.[16]

2.5 Multihoming

So far we have considered how to move existing communication flows between networks without disruption for ubiquitous communication, but not how to exploit any connectivity available. This solution to this is provided by ILNP by allowing transport flows to use multiple locators simultaneously. This is multihoming, which refers to connecting a node or network to more than one network.

ILNP supports this through sets of valid locators for an identifier and dynamic bindings of identifiers to locators.

The multipath effect refers to the effect of using multihoming - as there are multiple 'paths' packets can take. This can have connectivity and throughput benefits.

2.6 Security and Privacy

Network Address Translation (NAT) is a method of mapping between IP addresses spaces. It is useful in avoiding IPv4 address exhaustion, providing security benefits by hiding an internal private network, and hiding a node's location ensuring privacy. However, it requires writing IP source and destination addressed (and therefore checksums) and breaks the end-to-end principle.

ILNPv6 removes the need for NAT. 64-bit identifiers mean address space exhaustion is no longer an issue. To provide the security and privacy benefits of NAT Locator Rewriting Relays (LRRs) can be used. These are middleboxes that, as the name implies, rewrite locators and then forward the packets on.[6].

Ephemeral identifiers are another privacy measure that allows a node to use an identifier that only exists for the duration of a transport layer flow, preventing users from being tracked by ILNP identifier between sessions.

2.7 Other Mobility Solutions

As IP was not designed with mobility in mind most solutions try to retrofit mobility to IP. The middleware solutions previously discussed are one approach to this.

Other solutions such as MobileIP, Locator/ID Separation Protocol (LISP), and Host Identity Protocol (HIP) exist. However, all such solutions require either a proxy or agent, tunnelling, address mapping, or application modifications[9][16].

Contrasting this ILNPv6 requires modifications to the end hosts only. It is also unique in supporting a layer 3 soft handoff, as opposed to buffering packets through proxies, for example.

ILNPv6 does, however, require significant modifications to the kernel and a retooling of the network stack. These other solutions are easier to deploy as they do not require such fundamental changes. Although the lack of such fundamental changes is what results in their drawbacks.

3 Design

This chapter describes the design decisions made when creating the ILNP overlay network.

3.1 Network Stack and Overlay

The overlay network was created on top of UDP/IPv6. See figure 3.1. The layers labelled are the OSI model layers from the perspective of our overlay. Essentially UDP/IPv6 was treated as an unreliable link layer and our ILNP overlay network is built on top of that.

Note that while the ILNPv6 layer is a network protocol that uses an Identifier-Locator split it doesn't conform exactly to RFC6741[3] due to the peculiarities of implementing it in an overlay, and as there were features of ILNP that were not required by our experimental design, like locator rewriting relays.

Also, note the placement of the discovery protocol. It is most aptly placed in layer 3 as it is required for layer 3 communication. It's placed on top of ILNP as discovery messages are encapsulated in ILNP packets. However, it's not placed under STP, as STP packets are not encapsulated in discovery messages, but rather ILNP packets.

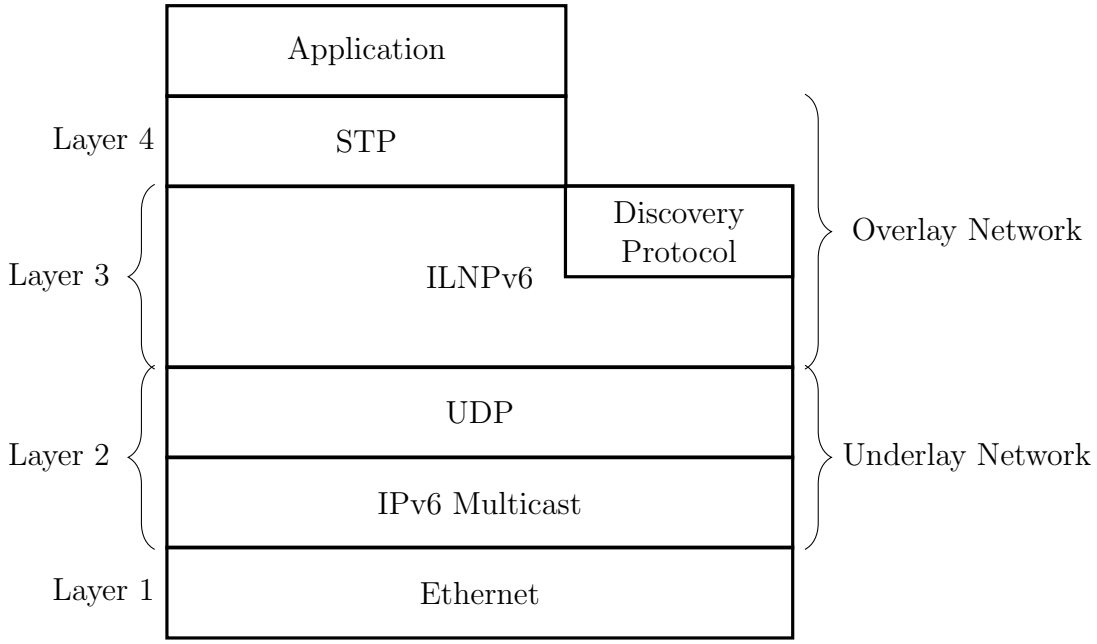


Figure 3.1: Overlay Network Stack

An IP multicast group corresponds to a virtual network. That is a network from our overlay network's perspective. Recall that a network is identified by a locator. Therefore a locator corresponds to an IP multicast address. All packets sent to and received from a locator in ILNPv6 are transmitted via the corresponding IPv6 multicast address with UDP. It is in this way we can create whatever virtual topology we desire whilst using machines connected locally communicating via IP multicast. From the overlay's perspective, IP multicast addresses are analogous to

MAC addresses identifying the interface on which to communicate. See figure 3.2 for the format of the IPv6 multicast group, and note that `Loc` denotes the corresponding locator.

The ILNPv6 header is as described in RFC6741[3]. Some fields, such as version, traffic class, and flow label, are unused. See figure 3.3.

The Skinny Transport Protocol (STP) simply provides a wrapper around ILNP packages with a source and destination port, similar to UDP. It doesn't provide checksums, as this is already implemented in UDP datagrams. See figure 3.4.

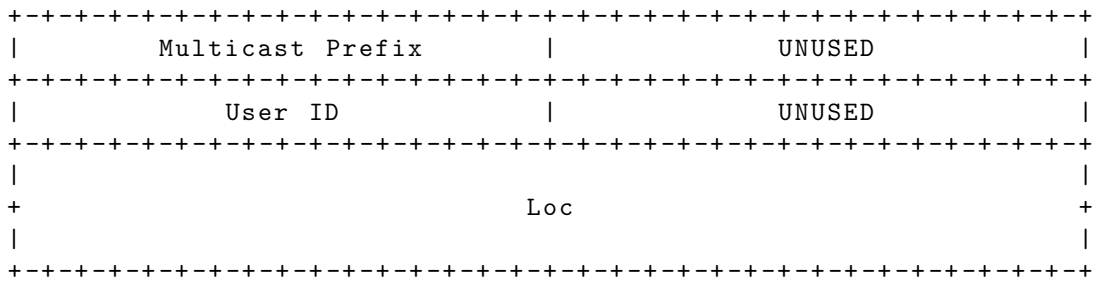


Figure 3.2: IPv6 multicast group address

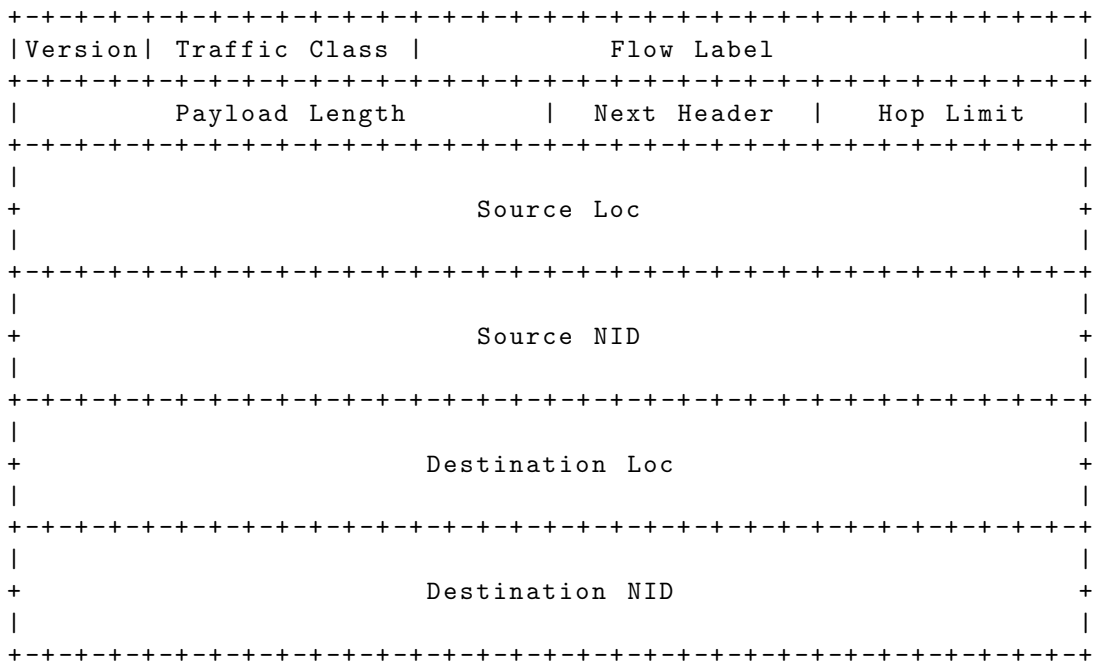


Figure 3.3: ILNPv6 header

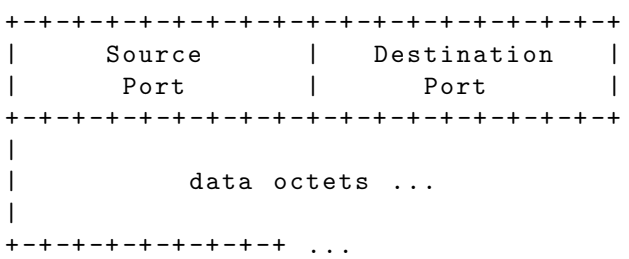


Figure 3.4: STP header

3.2 Discovery Protocol

A discovery protocol was required for nodes to find each other and to discover routing paths. The discovery protocol was modelled after IPv6 Neighbour Discovery Protocol[17], except with only Neighbour solicitations and advertisements, as there are no dedicated routers in our evaluation scenario. Every node can act as a router.

Also, a simple flood and backwards learn approach was taken as is common in WSNs. This means every node's solicitation/advertisement is forwarded to every other node in the network. When a node receives a discovery protocol message it forwards it to every interface except the one on which it was received. In a circular topology, this requires the hop limit to be decremented.

A solicitation informs receiving nodes of the sending node, whilst also requesting responses. Nodes respond to a solicitation with an advertisement, which informs the receiving node of the sending node. Advertisements are also sent to the special all nodes locator, on all interfaces. This means that nodes eavesdrop on advertisements, so one solicitation is sufficient for all nodes in a network to discover all the other nodes.

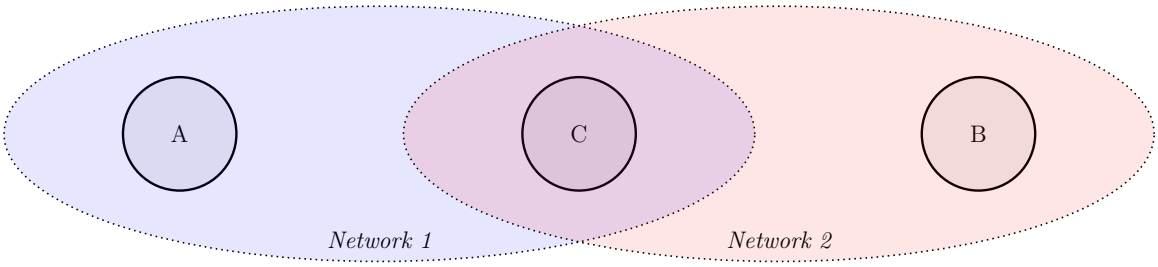
Both solicitations and advertisements contain a node's FQDN (hostname, in our network), and a set of valid locators. This means that domain name resolution is included in our discovery protocol, and DNS is not required. This is for the purposes of not requiring a DNS deployment in our overlay and simplifying the experimentation.

We shall demonstrate the operation of this protocol through an example illustrated with figures. Figure 3.5a describes the network topology of the example. Network membership is shown by overlaid coloured ellipses. The three nodes are connected in a chain with A in network 1, C in network 1 and 2, and B in network 2. Figure 3.5b shows a sequence diagram of the discovery messages sent over ILNP. At the end of the process, all nodes have received an advertisement or a solicitation from all other nodes.

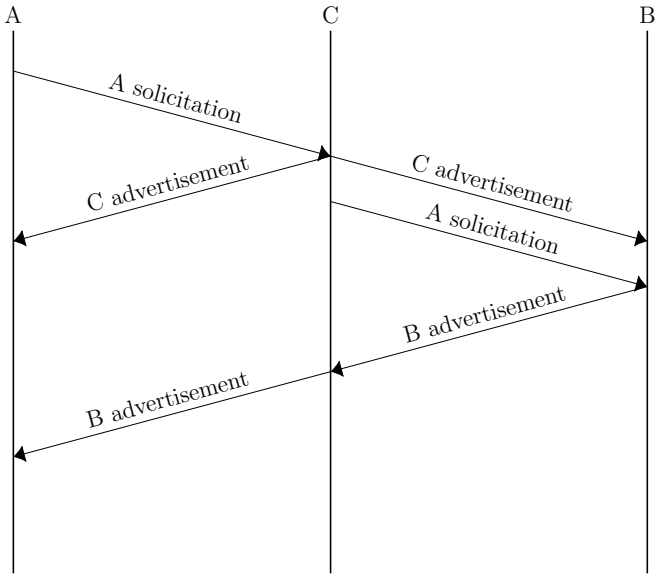
This protocol does not scale particularly well. For every node attached, the state per machine grows linearly, and the global state grows quadratically. For every new network added with a node, the number of advertisements sent in response to a solicitation grows quadratically. But it is sufficient for our use case.

Note that this discovery protocol is a control plane protocol, and is transparent to the user. The control plane is orthogonal to the network stack - that is, control plane protocols are present throughout all layers - but this discovery proctor sits in

the network layer as discussed in section 3.1. This is why the discovery protocol is shown as *not* underneath the transport and application layers in figure 3.1.



(a) Network Topology



(b) Sequence Diagram

Figure 3.5: Discovery Protocol Example

3.3 Mobility

The mobile node (MN) refers to the node that is currently moving networks, and the corresponding node (CN) refers to the node it is communicating with.

Mobility is provided through Locator Updates as discussed in 2.2. When a node updates its set of locators - when it moves network - it sends a Locator Update Message. This is sent from the MN via the current locator to the CN, the CN responds to the MN sending it to the MN's new locator, and when the MN receives it on the new locator the move is complete. This is covered in more detail in section 4.9.

Note that DNS is required to be updated with the nodes new locator. As domain name resolution is provided as part of our discovery protocol other nodes that do not have a current unicast session with the MN will receive its updated locator through the discovery protocol.

There is a soft handoff time during which the MN is connected to both locators. This allows any messages sent to the MN on the old locator to be received while

it transitions communication to the new locator. This is what provides the smooth handoff process.

3.4 Omitted functionality

LRRs were not added as they were not required for evaluative scenario.

Identifiers could have been created based on the MAC address of the interface configured for IP multicast (see section B.2) with duplicate address detection, like IPv6's SLAAC. But has decided this was over complicating the system for no benefit in the evaluative scenario, so identifiers for nodes were hard code.

Similarly, as identifiers were hard coded ephemeral identifiers were not implemented.

3.5 User Space & Python

As mentioned in the introduction, the focus of this project is on the protocol design and interaction. Because implementing networking code in the kernel poses significant technical challenges and would require time beyond that available for this project, the overlay network was implemented in user space. For similar reasons, Python was chosen as a language to implement it in. Python allows relatively fast development, as opposed to a language like C, at the cost of efficiency, as we will discuss in section 4.4. It also allows for very portable code that can be run on both x86 desktop workstations and ARM Raspberry Pis.

4 Implementation

This chapter describes the implementation details of the ILNP overlay network in Python.

4.1 Variable Naming

Identifiers are referred to as `nids`, locators as `locs`, and I-LVs as `ilvs`.

4.2 Multi-Threading and Concurrency

Some form of concurrency was required for the program. There is a requirement to process control messages at lower levels without waiting for the application to transfer control. For example, when the MN sends an ILNP Locator Update. There needs to be some logic for the network layer to received messages from the link layer without relying on the application to layer to return control. Multi-threading was used to provide this concurrency.

The Python Global Interpreter Lock (GIL) means that only one thread will ever execute at a time. This is nice to program with as it means we don't need to worry about a lot of threading issues. For example, basic data structures are thread safe in Python. But the disadvantage is there's no parallelism. If parallelism was required them either multiple Python processes with inter-process communication (IPC) would be required, or the use of a language that supports parallel threading. These multiple Python processes with IPC could also enable allowing multiple application layer programs to use the overlay network at once, potentially written in a language other than Python. Currently when an application layer program, like `heartbeat.py` or `experiment.py`, imports `transport.py` it instantiates the whole overlay network. This is not an issue for the purposes of this project as our evaluative scenario is only one application layer program that is not stress testing the network.

If our network thread that deals with Locator Updates checks for messages from the link layer but there are 5 data packets before the control Locator Update packet what is it to do? Due to this multi-threading buffers between layers are required. These are implemented as queues with Python `collections.deque`'s. It stores them in a buffer in `network.py` until the application/transport layer reads from it.

Thread synchronisation was added to the threads operating on these buffers to avoid busy waits, with Python `threading.Condition`'s.

In the transport layer this was done by *optionally* setting a bound socket to blocking. This is useful for the receiving nodes in the experiments as it allowed them to avoid a busy wait when reading from a socket. When a socket is set as blocking a conditional variable is created (mapped to by the socket's port) for synchronisation between the socket and the transport layer receiving thread. It is not possible to set a timeout value for socket reads like kernel sockets have, however, as this functionality was not required for our experiment.

We could have implemented a send buffer so that writing to a socket doesn't block if we can't send a packet immediately. Indeed this approach was taken but

then removed, for the following reasons. It made it unnecessarily complex to parse the logs for creating graphs in the experimental analysis. As packets were buffered before sending from the MN's application layer it wasn't known which interface packets were sent on. The CN's logs had to be cross-referenced by sequence number for each packet to determine which it was sent on. Additionally, with the UDP/IP multicast underlay network, it would have been a bit redundant, as UDP buffers in the kernel anyway.

4.3 Memory Management

This discussion of buffers brings us to the issue of memory management. Our implementation does not implement any bound on buffers. Unchecked they could grow to consume the entire machine's memory. To avoid this a bounded buffer size with a scheme like drop from head, drop from tail, or something more sophisticated like random early detection (RED), could be implemented. Alternatively, the write could just block until there's space in the buffer. This wasn't done as our evaluative scenario did not result in buffers using a significant amount of memory.

4.4 Python Efficiency

It's worth noting that Python is relatively inefficient, especially when compared to languages traditionally used for network protocol implementation, like C. It's especially inefficient when modifying a small part of a large packet, like decrementing the hop limit before forwarding, as it duplicated the whole packet to a new region of memory before making the modification. Again, this is not an issue for this project as our evaluative scenario is not stress testing throughput.

4.5 Module Structure

Figure 4.1 illustrates how the Python modules are organised based on their imports.

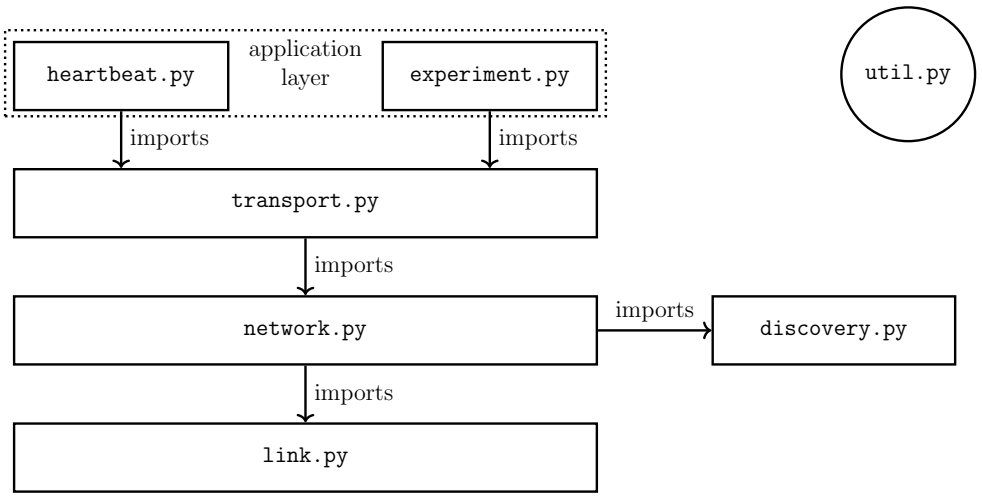


Figure 4.1: Module stack

The two application layer programs are `heartbeat.py` and `experiment.py`.

While the discovery protocol conceptually sits above the ILNP layer (see 3.1) - that is, its packets are encapsulated by ILNP - it is implemented as a module imported from the ILNP code. It was easier to implement it this way due to its integral role in the networking layer. While the discovery messages are encapsulated in ILNP packets, it would have added unnecessary complexity to implement it above the network layer like the transport layer. Recall from section 3.1 in figure 3.1 that it sits above ILNP, but still in layer 3. And from section 3.2 that it's a control plane protocol transparent to the higher layers.

The module `util.py` is imported by everything so is shown as a bubble. Some imports are skipped because they are not significant: `heartbeat.py` and `experiment.py` both import `discovery.py` but only to use the `gethostbyaddr` function.

Breakdown of functionality per module:

- `heartbeat.py`
  Used to test for the network. Sends a heartbeat between two nodes via a third acting as a router.
- `experiment.py`
  Used to perform the evaluative scenario test.
- `transport.py`
  The Skinny Transport Protocol layer.
- `network.py`
  The ILNP layer.
- `discovery.py`
  The discovery protocol logic and cache.
- `link.py`
  The UDP/IPv6 emulated data link layer.
- `util.py`
  Utility functions and logging functionality.

4.6 Transport Protocol

The transport protocol wraps outgoing packets with a source and destination port and buffers incoming packets by destination port. It does not support checksums or fragmentation. It supports blocking sockets as discussed in section 4.2.

4.7 Discovery Protocol

The discovery protocol described in section 3.2 was implemented as part of this overlay network. As discussed in section 4.5 this was done in `discovery.py` which was imported from `networking.py` as the discovery protocol is essential for layer 3 communication. This resulted in some logic of the discovery protocol being

implemented in `network.py` that requires being able to send ILNP packets. Otherwise `discovery.py` would have to import `network.py` and we could have a cyclic import.

An example of this functionality is `SolititationThread`. This thread sends a discovery protocol solicitation message if a message hasn't been received within the last `discovery.wait_time` seconds (see section B.2). Nodes eavesdrop on solicitations and advertisements. This means a discovery solicitation is only sent every `discovery.wait_time` in the entire network. There is the possibility of the threads synchronising across nodes, so all sent a solicitation message at the same. This could result in a detrimental amount of traffic in the network, especially considering the protocol already doesn't scale well. To address this the solicitation thread waits a random time from half the discovery wait time to the discovery wait time before checking when the last solicitation message was. This prevents solicitation thread synchronisation.

Note that the TTL of the mappings obtained from the discovery protocol is 3 times the `discovery.wait_time`. This means it will persist through 3 discovery messages being lost. This is a relatively arbitrary value and could be changed.

These discovery messages are sent to a special locator `ALL_NODES_LOC`, with a blank identifier. These are sent to a specific interface (a locator to which the node is connected). This is essentially a broadcast on a virtual network. The ILNP packets sent to this locator are not forwarded, but the packets are forwarded at the discovery protocol layer. This forwarding logic is not separated from the networking logic in the implementation as already discussed.

Backwards learning is done from the discovery messages. When a packet is received its source locator is mapped to the packet's received interface in the dictionary `loc_to_interface`. This dictionary is called the forwarding table. Outgoing packets then use these mappings to determine which interface to send a packet to reach the packet's destination locator. These discovery protocol messages act to 'jump start' the system.

4.8 Link Layer Emulation

As discussed in section 3.1 IP multicast addresses can be viewed as interfaces by our overlay network. The network layer isn't concerned with this representation, however, as there's a mechanical process to convert locators to IP multicast addressed in `link._get_mcast_grp`, see figure 3.2. For this reason, the conversion to IP addresses is done in `link.py`. In `network.py` when we refer to 'interfaces' we are in fact referring to locators to which the node is directly connected and can send and receive packets on the corresponding IP multicast address. These are stored in `locs_joined`. There is an additional check in `link.py` to make sure the node is connected to any multicast addresses it is sending to.

As a node receives all messages sent to an IP multicast group if it joins that group, the network layer filters out packets that have the receiving node's identifier as the destination locator, and don't have the `ALL_NODES_LOC` as the destination locator.

4.9 Mobility and Locator Updates

Locator updates are implemented to allow nodes to move locators, which correspond to virtual networks in our overlay. This requires keeping track of active I-LVs that have a current unicast communication session with the node. This is done through the `active_ilvs` dictionary that maps I-LVs to timestamps of the last unicast message received or sent to them. These mappings expire after `active_uncast_session_ttl` seconds. See section B.2 for configuration details.

In `network.py` there is a `MoveThread` that changes locator every `network.move_time` seconds, with a `network.handoff_time` second soft handoff period.

When a node (a MN) moves, it first joins it's new locator - or new set of locators (see section B.2) - and sends a discovery *advertisement* on the new locator(s). Note that this is not a solicitation so will not require a response. This is so that any nodes that can forward packets to the MN obtain a mapping from the MN's new locator to the locator's interface through backwards learning if the mapping doesn't already exist. The purpose of this message is path discovery through backwards learning, not node discovery, although it is a nice side effect that nodes not in a unicast communication session with the MN will also receive an update to its locator through a different mechanism than a Locator Update. In a more standard network, the CN would route the packets to the locator to the same uplink, with would then forward it on to the destination locator. Indeed our uplink for the CN, the router, does know how to forward packets to the MN's new locator because it has an interface directly to it. But as there were no nodes on the MN's new locator previously the CN does not know where to route packets to that locator. This is despite the fact that the CN only has one interface, but this isn't always guaranteed to be the case. Instead, the discovery message forwarded by the router to the CN provides path discovery so the CN knows which interface to send the packets to the MN's new locator to. This is why the discovery message is sent before the Locator Update - so the CN knows how to route packets to the new locator, before trying to after the Locator Update.

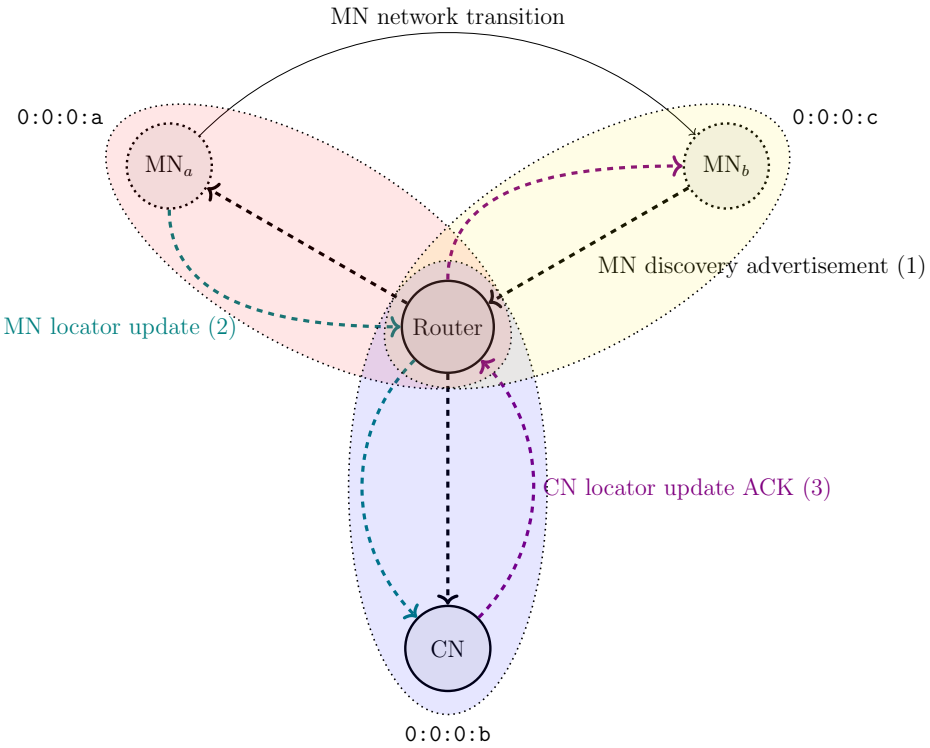
A discovery message is not required in the other direction, from the router to the MN, for the MN to perform backwards learning on. This is because the Locator Update acknowledgement is sent from the CN to the MN on the new locator, so the MN can perform backwards learning on this to map the CN's locator to its new interface. Note that the MN will not be able to send packets to other locators until the next discovery protocol solicitation message.

The node is restricted from sending messages on the old locator after it has joined the new locator, but it can still receive messages on the old locator for the duration of the soft handoff. This is what provides the smooth transition and lack of loss.

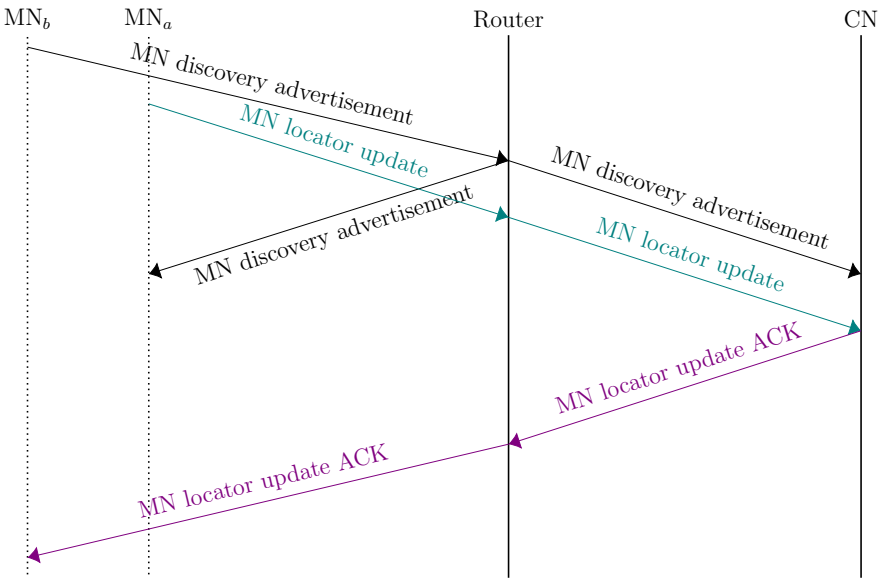
The MN then sends a Locator Update message to all nodes in a current unicast communication session with it. The Locator Update message is sent via the old set of locators as determined by the forwarding table. When the CN receives this it updates its hostname to I-LV mappings in `discovery.py` and responds with a Locator Update acknowledgement. The MN will retry a Locator Update up to 3 times by default. The MN will leave the old locator, or set of locators, after the soft handoff period. The forwarding table is then reset by removing any mappings to interfaces that the node is no longer connected to.

Locator updates contain the new set of valid locators for the node that sent them. While RFC6743[1] specifies a preference to be transmitted for each locator this was not done in our implementation as we do not consider a multihoming policy in our experiment. When resolving a hostname to an I-LV if multiple locators are available the first one is chosen.

We shall demonstrate the operation of this protocol with an example from the experiment testbed, illustrated with figures. Figure 4.2a shows the network topology, the mobile node's network transition from `0:0:0:a` to `0:0:0:b`, and the networks the messages are sent in. Note that MN_a refers to MN in locator `0:0:0:a` and MN_b to the MN in `0:0:0:a`. Figure 4.2b shows the timeline of these messages. Note the colour coding corresponds to figure 4.2a. The MN first sends out a discovery message for the CN to discover a route to `0:0:0:b` through backwards learning. As a side effect, the MN also receives its own discovery message on `0:0:0:a`. The MN then sends a Locator Update on `0:0:0:a`, and the CN response with a Locator Update acknowledgement on `0:0:0:b`.



(a) Network Topology



(b) Sequence Diagram

Figure 4.2: Locator Update Example

5 Experiments

This chapter describes the experiment used for the evaluative scenario and the results of this experiment.

5.1 Methodology

5.1.1 Testbed

The testbed consisted of 3 Raspberry Pis with cases and power cables, 1 Ethernet switch and power cable, and 3 Ethernet cables. See figure 5.1 for a picture of the physical devices and networking equipment used.

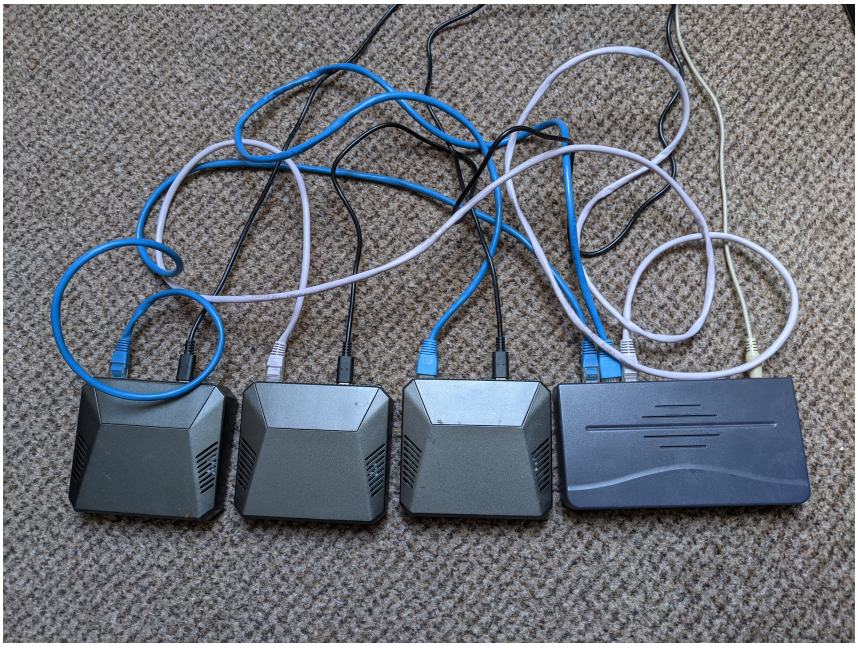


Figure 5.1: Physical Testbed

The versions of the software being used are Ubuntu Server LTS 20.04.2, Linux Kernel 5.4.0, and Python 3.8.5. The Raspberry Pi's are Raspberry Pi 4 Model B Rev 1.4's.

The inbuilt WiFi adapters of the Pis were used as a control plane to avoid interfering with the experiments running over Ethernet.

They were configured with static IPv6 addresses on their Ethernet interfaces. See section B.5 for details.

5.1.2 Network Topology

One Pi acts as the CN, one as the MN, and one as a router connected to all three networks simultaneously. This one router is emulating a number of routers that would make up the connection between networks. The Pis were named for easy `ssh` access and configuration. Alice was used as the MN, Bob as the CN, and Clare as the router.

The network topology, emulated via IP multicast groups as discussed in section 3.1, consists of 3 networks. The CN resides at one of these networks and the MN moves between them every 20 seconds. There is a 10 second soft handoff period during which the MN is connected to both the old and new locators.

See figure 5.2 for a diagram of the network topology described above. The red, blue, and yellow ellipses show node network membership of `0:0:0:a`, `0:0:0:b`, and `0:0:0:c` respectively. The MN moves between the locations shown every 10 seconds in a cycle, starting at MN_0. The MN is shown in the overlap between the networks during the soft handoff period. The dashed lines show the start of the soft handoffs, and the solid lines the end of the soft handoff. A dashed and solid line together make up a network transition. The CN is in locator `0:0:0:b` for the duration of the experiment. The Router is in all locators for the duration of the experiment.

This meets the objects of the experiment by emulating a highly mobile IoT communicating with a static IoT device. Note that if we were emulating an IoT device communicating with a server then we might not make the CN reside at one of the networks the MN visits.

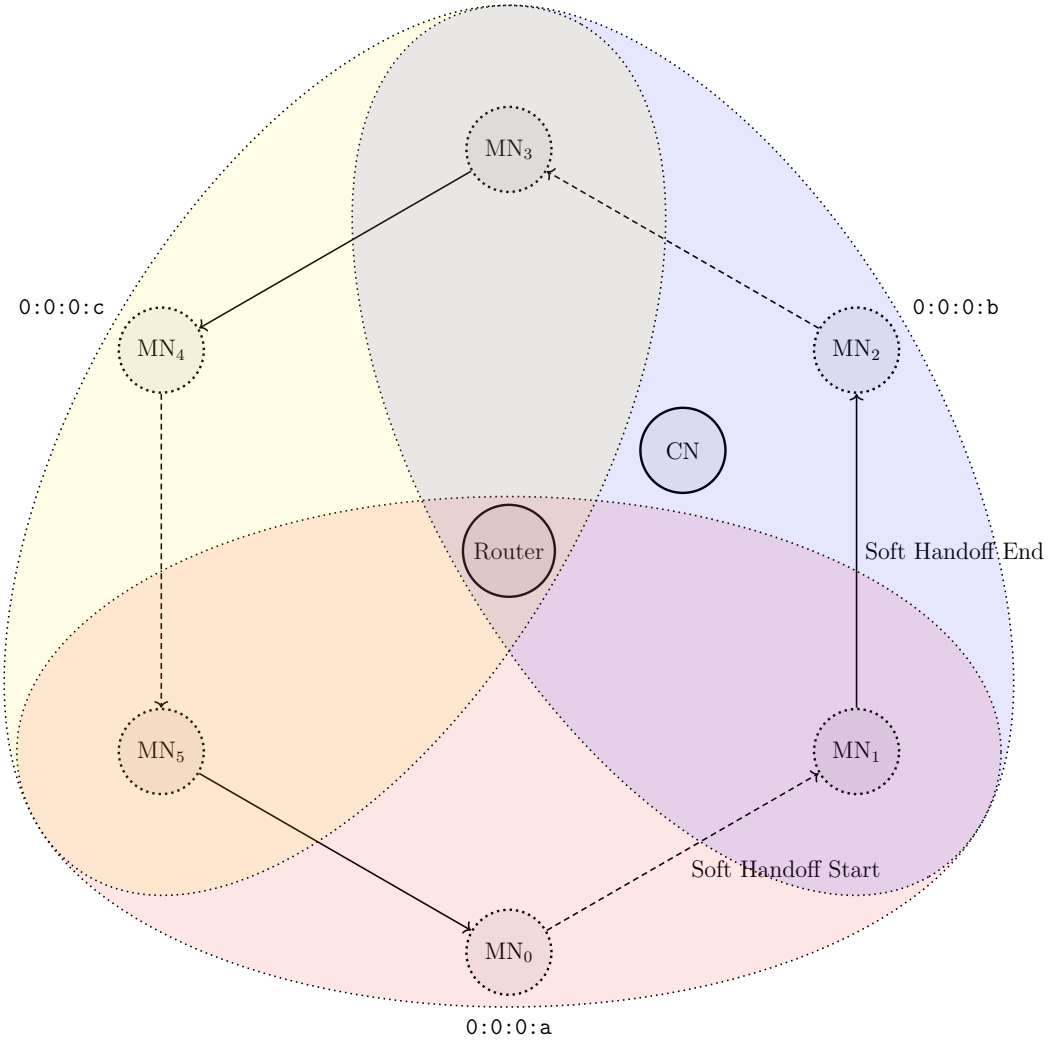


Figure 5.2: Experiment Virtual Topology

There will be an amplification effect due to the IP multicast emulation whenever

the MN doesn't reside at the same network as the CN. For every 1 packet sent in the overlay network, 2 IP multicast packets will be sent: one for the MN to the router, and one for the router to the CN.

5.1.3 Experiment Application

The experimental application is implemented in `experiment.py` It sends a packet of random bytes the size of the MTU every 10ms to a remote node if specified. The is the MTU of UDP minus the headers used for overlay network heads, resulting in an MTU of $1440 - 44 - 8 = 1388$ bytes.

Scripts used to automate the experimental procedure including deploying code, deleting remote logs, running processes, and retrieving logs. See B.4.

5.1.4 Configuration

The configuration values used were:

- `link.mcast_port = 10000`
  As this was an unused port on the Pis.
- `link.mcast_interface = eth0`
  As this is the Pi's name for the Ethernet interface.
- `link.buffer_size = 1440`
  As the UDP MTU is 1440 bytes.
- Identifiers were statically assigned to be unique and differentiable from locators at a glance with a `ffff` prefix.
- Locators were assigned according to the network topology as described in figure 5.2.
- `network.default_hop_limit = 3`
  As the furthest a packet will have to travel in this topology is one hop, 3 is more than enough.
- `discovery.wait_time = 30`
  Discovery messages every 30 seconds is a sensible default that will allow nodes to remain up to date of the network while not impacting the experiment's throughput.
- `network.backwards_learning_ttl = 30`
  30 seconds is a sensible default, as the discovery solicitation messages are sent every 30 seconds. As the experiment runs with a constant stream of packets this value won't really matter as long..
- `network.active_unicast_session_ttl = 30`
  For the same reasoning as `network.backwards_learning_ttl`.
- `network.loc_update_retry_wait_time = 1`
  As the RTT from the CN to MN in this topology is well under 1 second.

- `network.loc_update_retries = 3`
  This is a sensible default. The chance of 3 packets being lost over a time period of 3 seconds is very unlikely, and if it does occur we assume either the network or the remote host is down.

- `network.move_sleep = 20`
  As this is short enough for a highly mobile device, but long enough to show stable throughput in between moves.

- `network.handoff_time = 10`
  To allow plenty of time for any packets via the old locator to reach the MN. This could probably be reduced without impacting the experiment but it results in a nice consistent network change every 10 seconds with the `network.move_sleep = 20`.

- Logging was enabled for all layers for parsing the experiment results and debugging.

See section B.2 for details of what these configuration values mean.

The code in `data_processing/process.py` was used to create the following graphs using `matplotlib` from the application layer experiment log files.

5.2 Results

Three experiments were ran. Experiment 1 was a unidirectional flow from the CN the MN (figures 5.3 and 5.6), experiment 2 was a unidirectional flow from the MN the CN (figures 5.4 and 5.7), and experiment 3 was a bidirectional flow from the CN the MN and vice versa (figures 5.5a, 5.5b and 5.8).

For experiments 1 and 2, packets of 1388 bytes are sent every 10ms, so 100 packets are sent in a 1 second bucket, and $1388 * 100 = 138.8$kB/s is the expected throughput. Experiment 3 has two flows so we expect a throughput double this, which is $138.8 * 2 = 277.6$kB/s. These data rates were chosen to test the system with a constant stream of packets, but not enough to overload it.

Flows of 500 seconds were done for each experiment.

There are two types of graphs included. The first is the received sequence numbers vs time on a node. These display the sequence numbers received by the specified node plotted against the time they were received. The time during which the node moves locator is shown with vertical dotted lines. The soft handoff ends 10 seconds after the move, halfway in between these lines, as the moves happen every 20 seconds. See figures 5.3, 5.4, 5.5a, and 5.5b.

The second type is the throughputs in 1 second buckets vs time. These show the throughput grouped in 1 second buckets plotted against time, by locator, as well as the aggregate throughput across all locators. The locators refer to the locator in use by the MN, even when measuring throughput on the CN where they refer to the locator the packet was sent/received on.

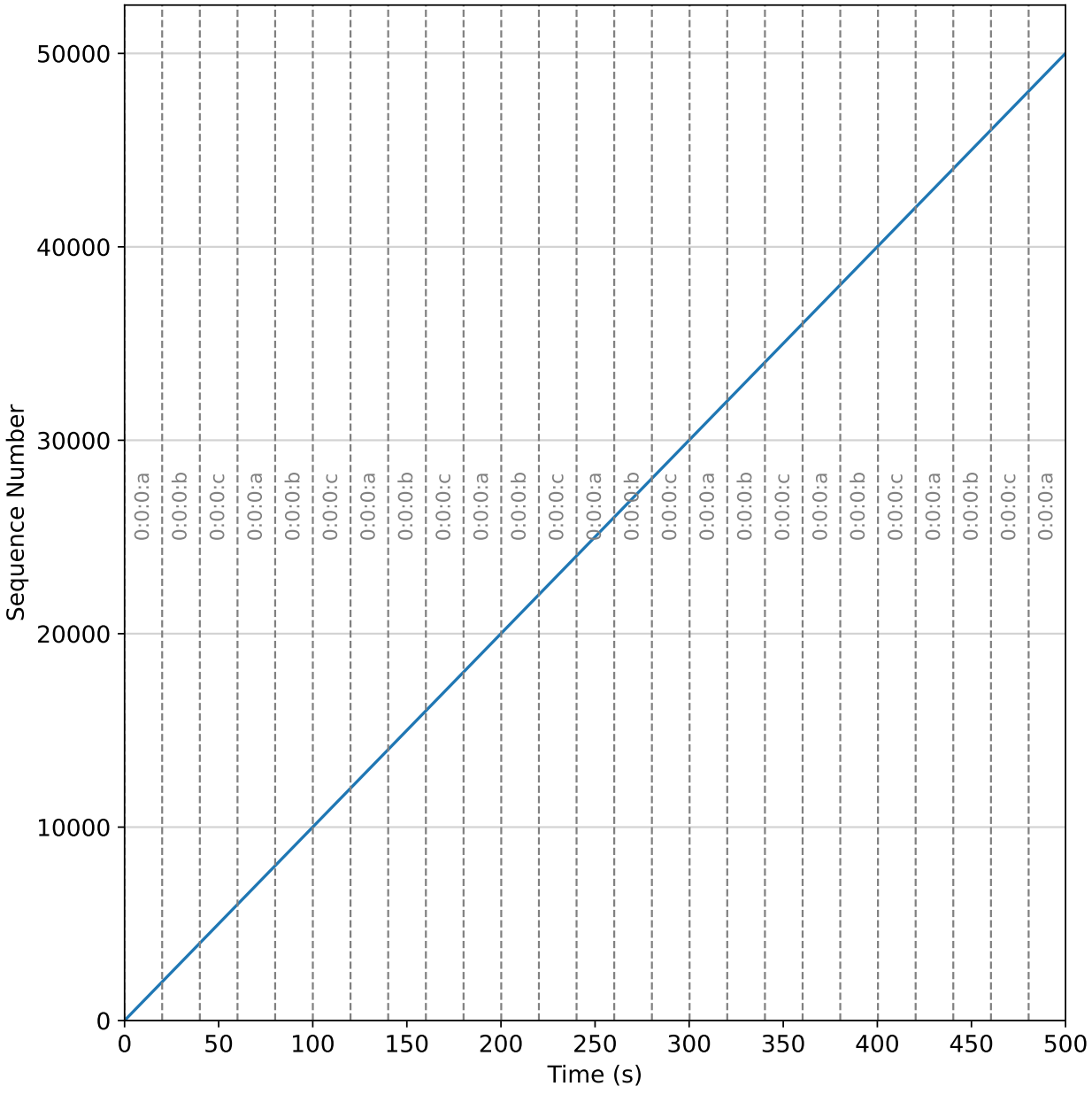


Figure 5.3: Experiment 1 CN->MN
Received sequence numbers vs Time on MN

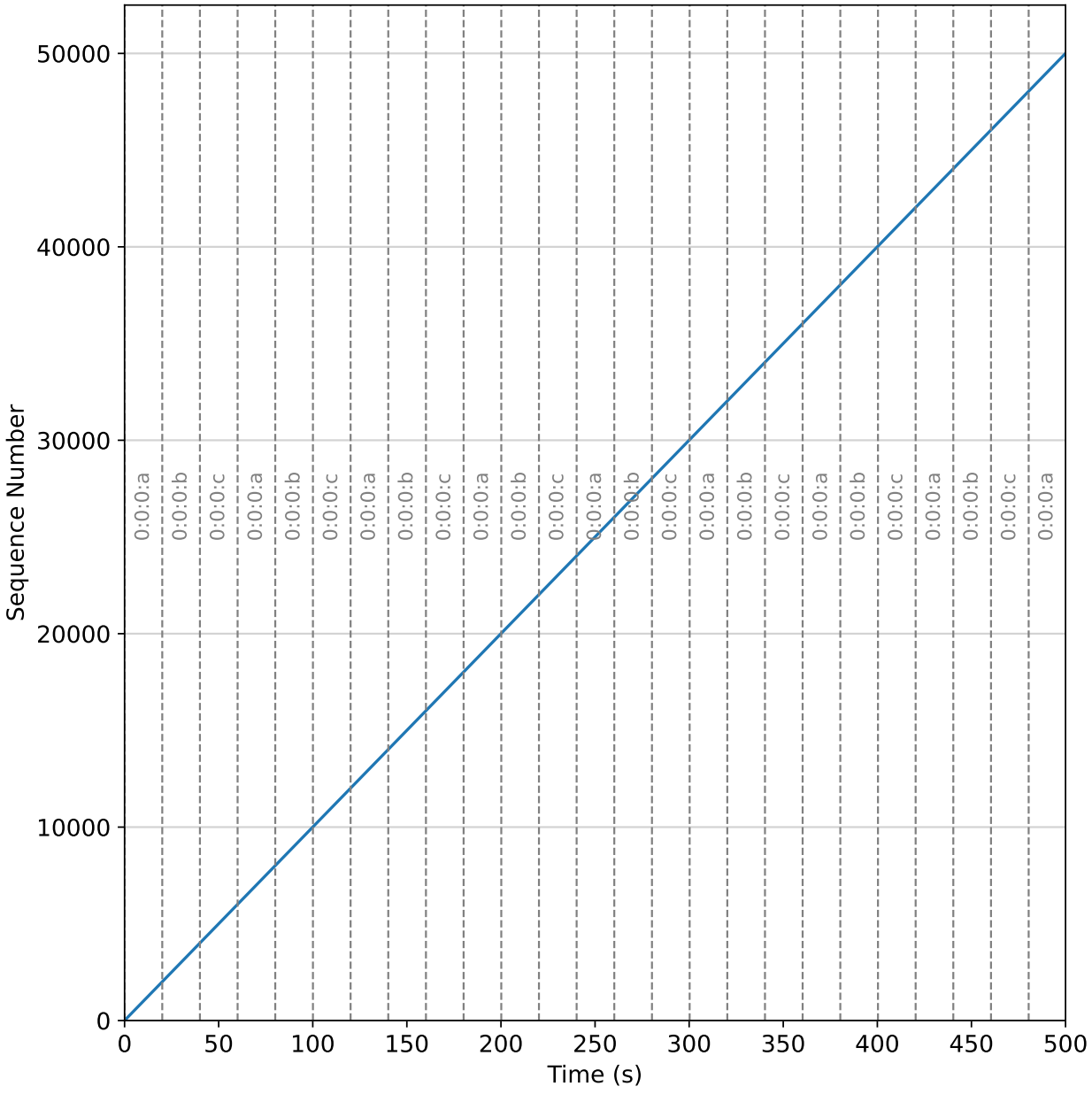
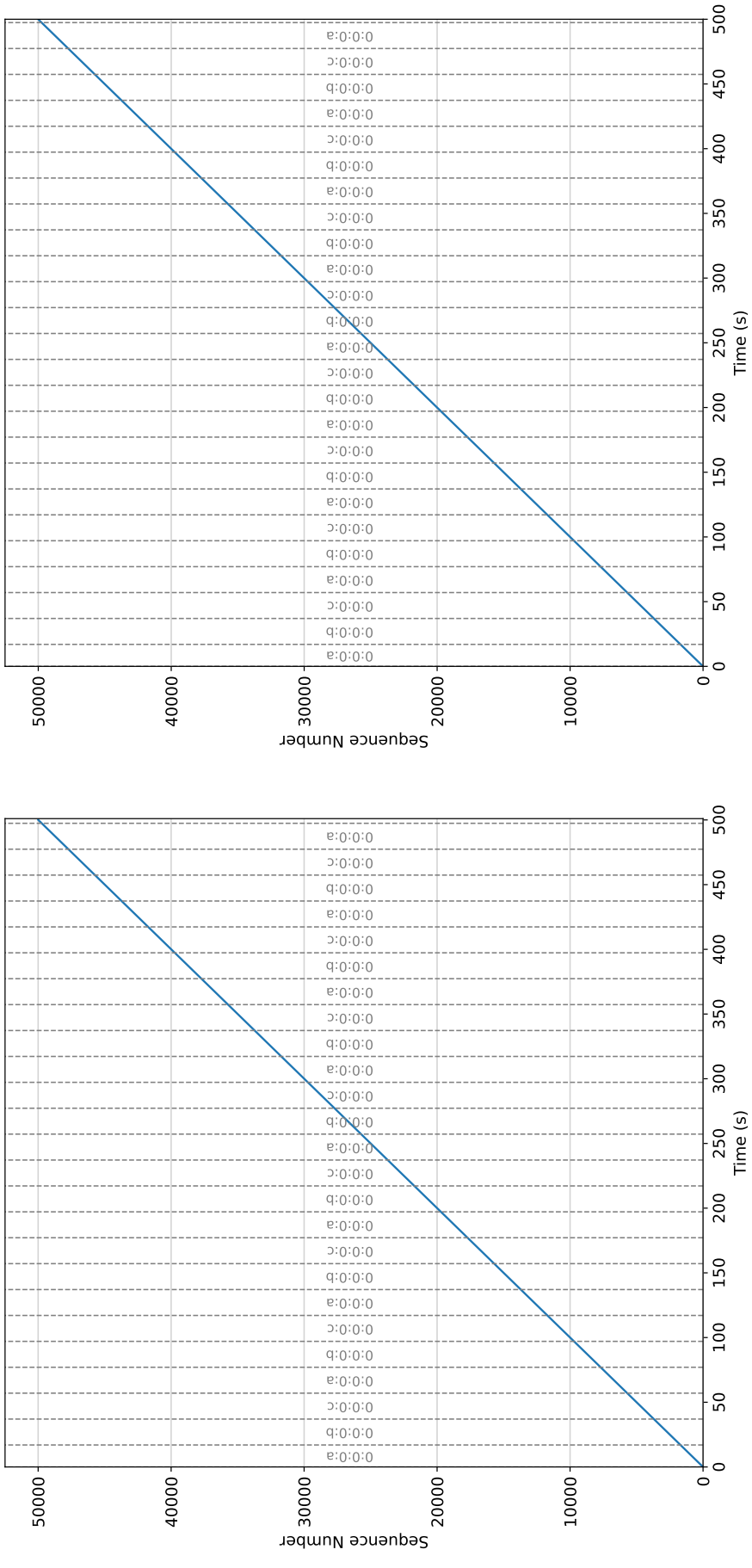


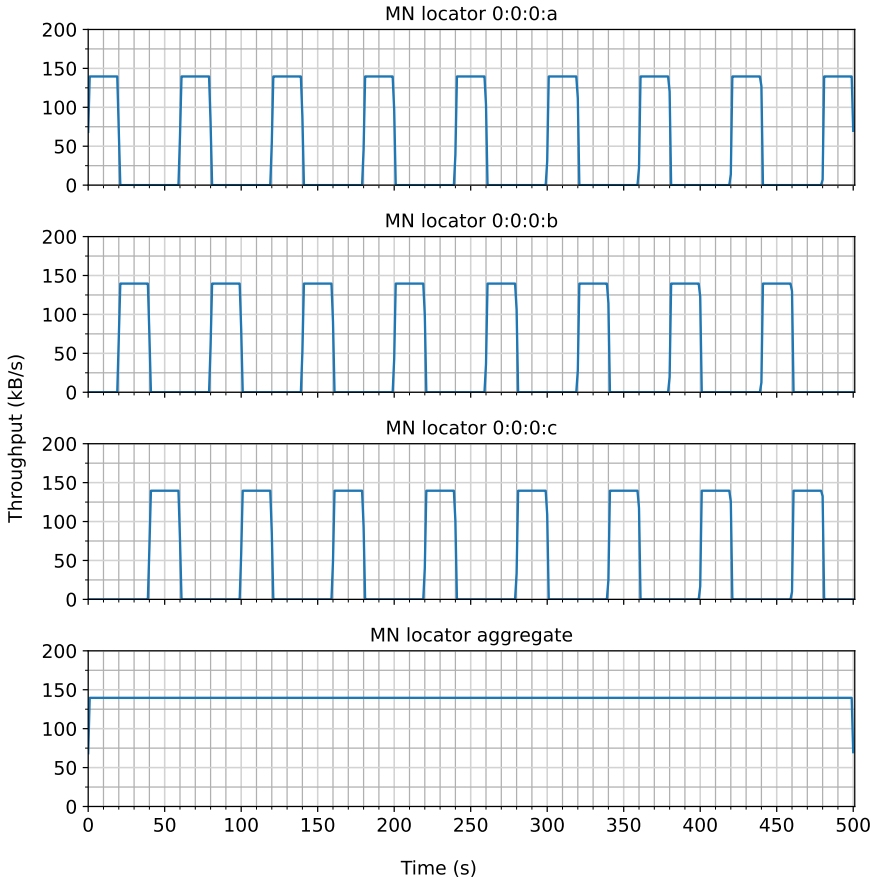
Figure 5.4: Experiment 2 CN<-MN
Received sequence numbers vs Time on CN



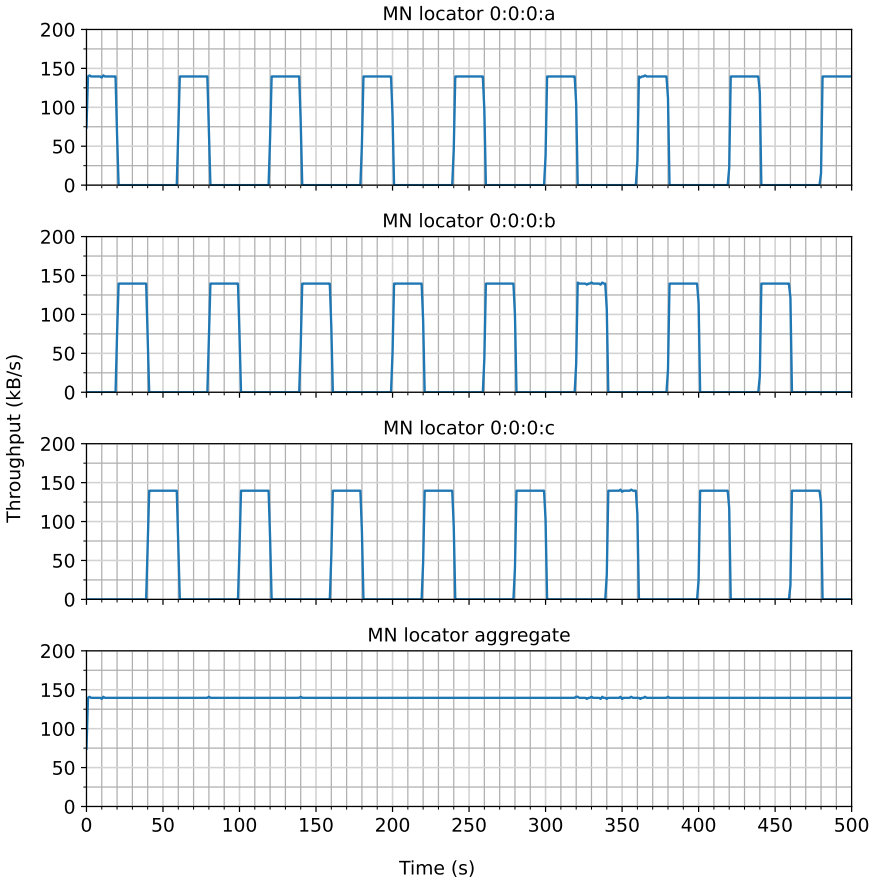
(a) Received sequence numbers vs Time on MN

(b) Received sequence numbers vs Time on CN

Figure 5.5: Experiment 3 CN<->MN
Received sequence numbers vs Time

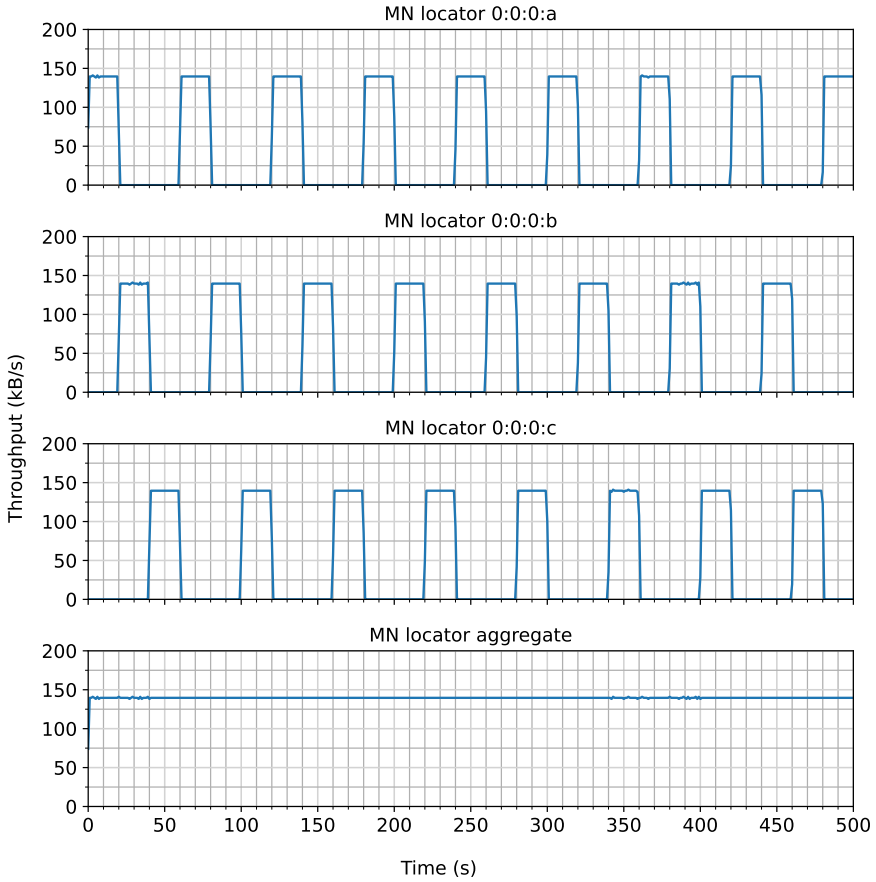


(a) Throughput in 1s buckets vs Time on CN

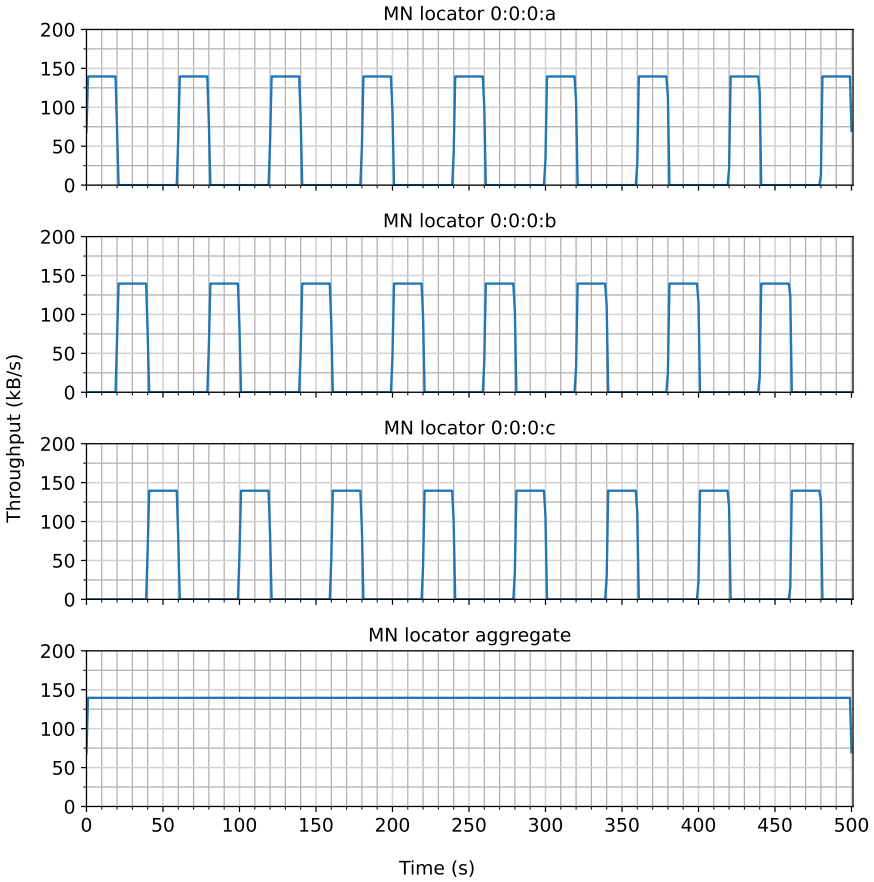


(b) Throughput in 1s buckets vs Time on MN

Figure 5.6: Experiment 1 CN->MN
Throughputs in 1s buckets vs Time

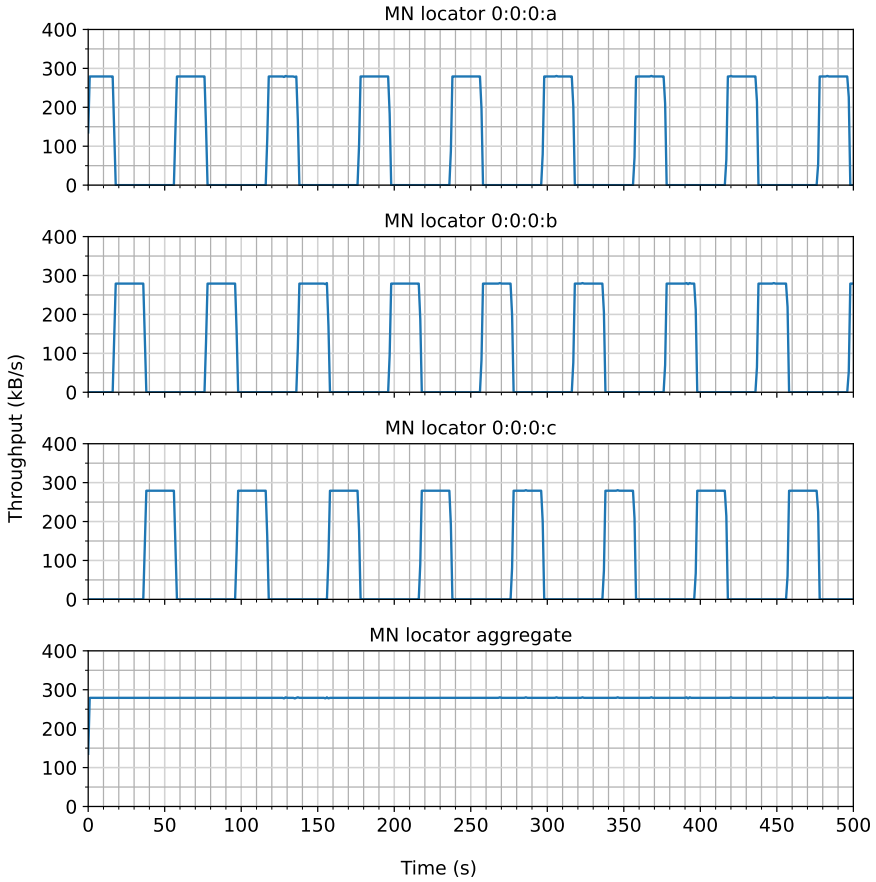


(a) Throughput in 1s buckets vs Time on CN

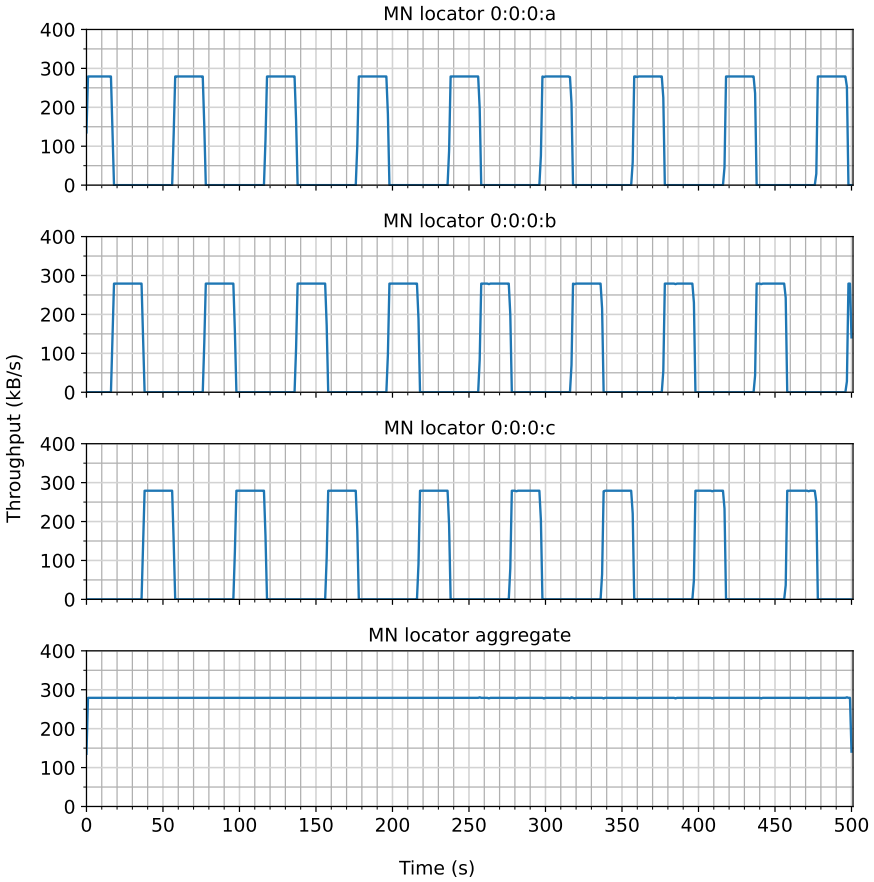


(b) Throughput in 1s buckets vs Time on MN

Figure 5.7: Experiment 2 MN<-CN
Throughputs in 1s buckets vs Time



(a) Throughput in 1s buckets vs Time on CN



(b) Throughput in 1s buckets vs Time on MN

Figure 5.8: Experiment 3 MN<->CN
Throughputs in 1s buckets vs Time

5.3 Analysis and Discussion

The sequence number graphs give us the overall picture and show us that there is no loss or misordering of packets as there are no vertical or horizontal breaks in the graph. The trend is a smooth straight line, which shows seamless connectivity is maintained through network transitions, including the end of the soft handoff that occurs halfway in between the locator moves shown by dotted vertical lines. Note that as a packet is sent every 10ms, i.e. 100 packets in a second, for 500 seconds, 50000 packets are sent.

The throughput graphs show us more detail. The discrete rectangles in the per locator throughput graphs show a distinct separation between locator uses. There is a slight gradient to some of the rectangle's vertical edges. This is due to the grouping of packets into 1 second buckets, when part of the bucket captures packets transmitted via one locator, and part of the bucket via another locator.

The smooth aggregate throughput shows seamless connectivity across locator moves. The sending node's throughput is useful to see how the node is sending to the network, but the receiving node's throughput is more interesting as it shows the results of the packets travelling through the network.

There is a slight wobble in the first rectangles of the throughput on locator `0:0:0:a` and `0:0:0:b` in figure 5.7a, experiment 2's throughput on the receiving CN. This is likely a small systems scheduling or buffering issue, due to another running process, but is not significant.

The throughputs match the expected values of 138.8kB/s for experiments 1 and 2, and 277.6kB/s for experiment 3.

Overall, we can see that our protocol has successfully provided layer 3 mobility in our overlay network with a seamless transition of existing communication flows through the use of a soft handoff.

6 Evaluation

Due to UDP multicast requiring all receivers of a packet to be listening on the same port, it is not possible to run multiple instances of the program on the same machine. A way around this would be sending packets on multiple ports for each instance of the program, but this scales linearly and defeats the point of using multicast. Having this option would, however, allow more sophisticated testing of higher layer mechanisms like the discovery protocol with more complex network topologies, and Locator Updates across multiple network hops.

Despite the discussion around the disadvantages of middleware as the network was implemented in an overlay network, we have essentially implemented our own form of middleware. There are Python interpreters available for IoT and embedded devices, but the whole purpose of using ILNP is to provide mobility at layer 3 and move aware from middleware. The underlay network, UDP/IP multicast emulating an unreliable link layer, could be replaced with another packet switching technology, such as IEEE 802.15.4/Zigbee, or IEEE 802.11/WiFi, with some low-level link layer emulation code like that in `link.py` for UDP. This could require reimplementing buffering for outgoing packets with a send queue. Doing this would remove the middleware aspect of the program, but there are still numerous issues with this implementation for real-world use, as discussed in section 4.

However, the purpose of this project has not been to implement a performant network stack applicable to real-world use cases, but rather to provide a 'proof of concept' demonstrating the operation of the ILNP protocol and how it enables mobility. Our evaluative scenario has demonstrated this, in that a layer 3 protocol can successfully provide layer 3 mobility. If ILNPv6 was implemented in the network stack, in the kernel, then it would provide native mobility support in the network layer. This has been shown to be possible with an experimental ILNP implementation in the v4.9 LTS Linux kernel and an experimental analysis on workstation PCs in 'Seamless Internet connectivity for ubiquitous communication'.[9] Previous work in this area has been limited to experiments on workstation PCs and server machines. This project has demonstrated that the ILNP protocol support mobility on lower power IoT devices, Raspberry Pis, as they are more resource-constrained than workstations and servers. Further work could include experimenting on even more resource-constrained, and IoT-like, devices.

When proposing an alternative solution to a problem it makes sense to compare it to the alternatives. The most obvious solution here would be MobileIP. But this proved problematic here as a whole other MobileIP overlay network for the comparison to be fair due to the performance restrictions of programming in an overlay. Even setting up a non-overlay MobileIP deployment would have been very time-consuming. Instead, we compare our results to those from 'Seamless Internet connectivity for ubiquitous communication'[9]. This paper has compared ILNPv6 in the kernel to MobileIP, on workstation PCs, and shown the performance and connectivity benefits ILNP has. We have reproduced the ILNP results of this paper, except in an overlay and on IoT devices: the Raspberry Pis. This shows that the same benefits of ILNP translate to a more resource-constrained environment.

The original plan was to perform the experimental analysis with WiFi, as a

wireless link layer technology is more appropriate to an IoT context, but issues described in section C resulted in changing to use Ethernet as the link layer technology. This did result in more reproducible results, however, as WiFi has a much more variable link association time.

We have analysed the scenario of one static CN communicating with a mobile MN in chapter 5. Another viable scenario is two mobile nodes communicating with each other. This would be emulating two mobile IoT devices communicating directly with each other.

The overlay's Skinny Transport Protocol is a stateless protocol. It's similar to UDP in that it simply wraps application layer packets with a port for multiplexing and passes it to the network layer. While the lack of loss we see during layer 3 transitions in the experimental results is very useful, the transparent switching of locators is even more beneficial to protocols, and applications, that have more state. This was discussed in chapter 2. Further work would include experimenting with protocols and applications that rely on more state and have a more complex and resource-intensive setup procedure. The Transmission Control Protocol (TCP) is one such example, as it relies on a 3-way handshake, keeps track of sequence numbers, and has a state machine. Transport Layer Security (TLS) is another example that has even more state and setup procedure, due to the cryptographic key exchange.

This project has been concerned with how to move between networks without disruption to communication flows, but as discussed in section 2.5 exploiting any and all connectivity available is another important facet of Ubiquitous communication. The soft handoff can be thought of as a temporal form of restricted multihoming. Temporal in the sense that its duration is limited to the duration of the handoff process, and restricted in that packets can be received on the old interface but not sent to it. Through the experiment's results, we can see that the MN is successfully multihoming during the soft handoff. Indeed this is what allows for such a smooth handover process without loss. We only experiment with transitioning from 1 old network to 1 new network, however, so are limited to multihoming on two networks. Also, there is no implementation of a multihoming policy. This would require changes like preferences in Locator Updates and a policy to choose locator when resolving an I-LV. Currently when there are a set of valid locators available the first one is simply chosen, as described in section 4.9.

To discuss our scenario from section 1, the layer 3 mobility provided by ILNPv6 implemented in the kernel would provide the required agility for the BAN allowing it to seamlessly transition between layer 3 networks. This seamless transition is especially important for these resource-constrained devices. The multihoming support provided by ILNP would allow it to make use of any and all connectivity available. The advantage of this over approaches like MobileIP is the increased connectivity performance through a seamless transition and lack of buffering in proxies.[9] The advantage over middleware is that it doesn't tie the application developers of the health monitoring devices to a specific middleware vendor, or if they do still choose to use middleware it allows a skinnier middleware. The possible issues with this approach are that it's much harder to deploy an ILNPv6 implementation than the alternatives.

To summarise how to original objectives of the project were met:

1. Objective P1 was met with the provided source code for the overlay network.
2. Objectives P2 and T1 were met with the evaluative scenario and experiment on the Raspberry Pis testbed.
3. Objective P3 was met with section 2.6.
4. Objective S1 was partially met by analysing the temporal multihoming that is the soft handoff.
5. Objectives S2 and T2 were not met due to time constraints.

See section 1.1 for a list of objectives and their codes.

7 Conclusions

This project involved implementing an ILNP overlay network and performing an experimental analysis in an IoT context to demonstrate the mobility supported by the protocol, which enables Weiser's vision of Ubiquitous Computing. This builds on previous work in this area by analysing the performance of ILNP in an IoT scenario with resource-constrained devices, in contrast with the workstation PCs and server machines used in previous work. Even though an overlay network has been built, the concepts and protocol operation translate into kernel code.[9]

The key achievements of this project have been successfully designing and implementing an ILNP overlay network, focusing on protocol design and operation; obtaining experimental results from an evaluative scenario based on an IoT setting with resource-constrained devices; and demonstrating ILNP's seamless layer 3 transitions through the use of a soft handoff. Overall, the main priorities of the project were able to be completed and the experimental results demonstrate what was anticipated.

The limitations of this work are the performance of the program due to the overlay and use of Python; the scaling of the discovery protocol; only one application program is supported for a virtual network stack as it runs on a single process without IPC; and only one instance of the program can be run on a machine, due to the multicast UDP socket used by each instance of the program being bound to the same port.

Further work in this area includes experimenting with a kernel implementation of ILNPv6 on IoT devices; investigating a multihoming policy and the benefits gained from the multipath effect for IoT devices; performing experiments of IoT devices transitioning between networks using a wireless communication link layer such as IEEE 802.11/WiFi, as this more appropriate than Ethernet for an IoT context; performing experiments with two mobile nodes communicating; and performing experiments with even more resource-constrained devices than Raspberry Pis, such as wireless sensors nodes.

A Testing

A.1 Debugging

Due to the nature of the operation of this program, most debugging was done by either looking at the live logs with `ssh` and `tail -f`, or by analysing logs statically after a run.

A.2 Unit testing

Unit tests are not provided for this project. This is partly due to the focus on the experimental aspect of the project over software engineering, but also the fact that network code is not suited to unit testing. There is no testing framework for a distributed system. Rather the program was tested as a system as the overlay was built from the ground up. This included testing, for example, the encoding and decoding of packets in each layer.

A.3 Integration testing

Integration testing was done with the heartbeat program in a variety of network topologies, as described in section A.4.

A.4 Discovery Protocol

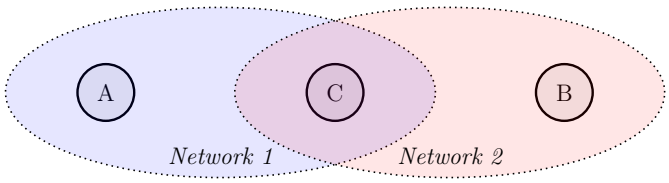
The discovery protocol was tested with a number of different network topologies, but these were limited by the number of physical devices available to test on: 3 Raspberry Pis.

The chain topology in figure A.1a tests the discovery message forwarding works from node A to node B via node C, and vice versa. The ring topology in figure A.1b tests the hop limit decrementing prevents discovery messages from being forwarded forever. The local topology in figure A.1c tests the protocol works while all the nodes are in the same network.

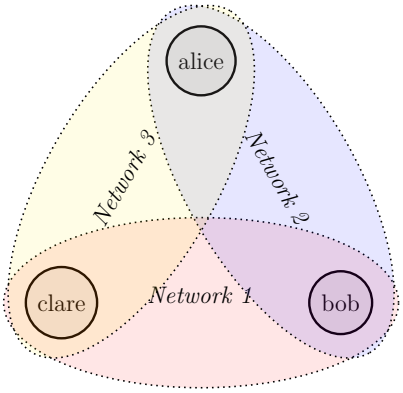
A.5 Soft Handoff Bug

An example bug, its detection, and its solution, is described below.

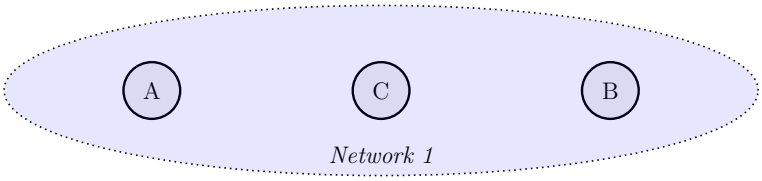
During the experimental stage when the MN moved from the same network as the CN, the throughput doubled for the duration of the soft handoff. After some investigation this was revealed to be due to packets being received twice: both on the old locator where the packet originated from and the MN was still connected to, and on the new locator where the packet was destined for where the MN had just connected to.



(a) Chain Topology



(b) Ring Topology



(c) Local Topology

Figure A.1: Testing Topologies

This was fixed by the following code in `network._receive`:

```
# If for us, but received on a locator that we're not
   ↪ currently joined to, ignore.
# This is required for not receiving duplicate messages
   ↪ during the soft handoff.
if received_interface not in locs_joined and dst_loc !=
   ↪ recieved_interface:
    return
```

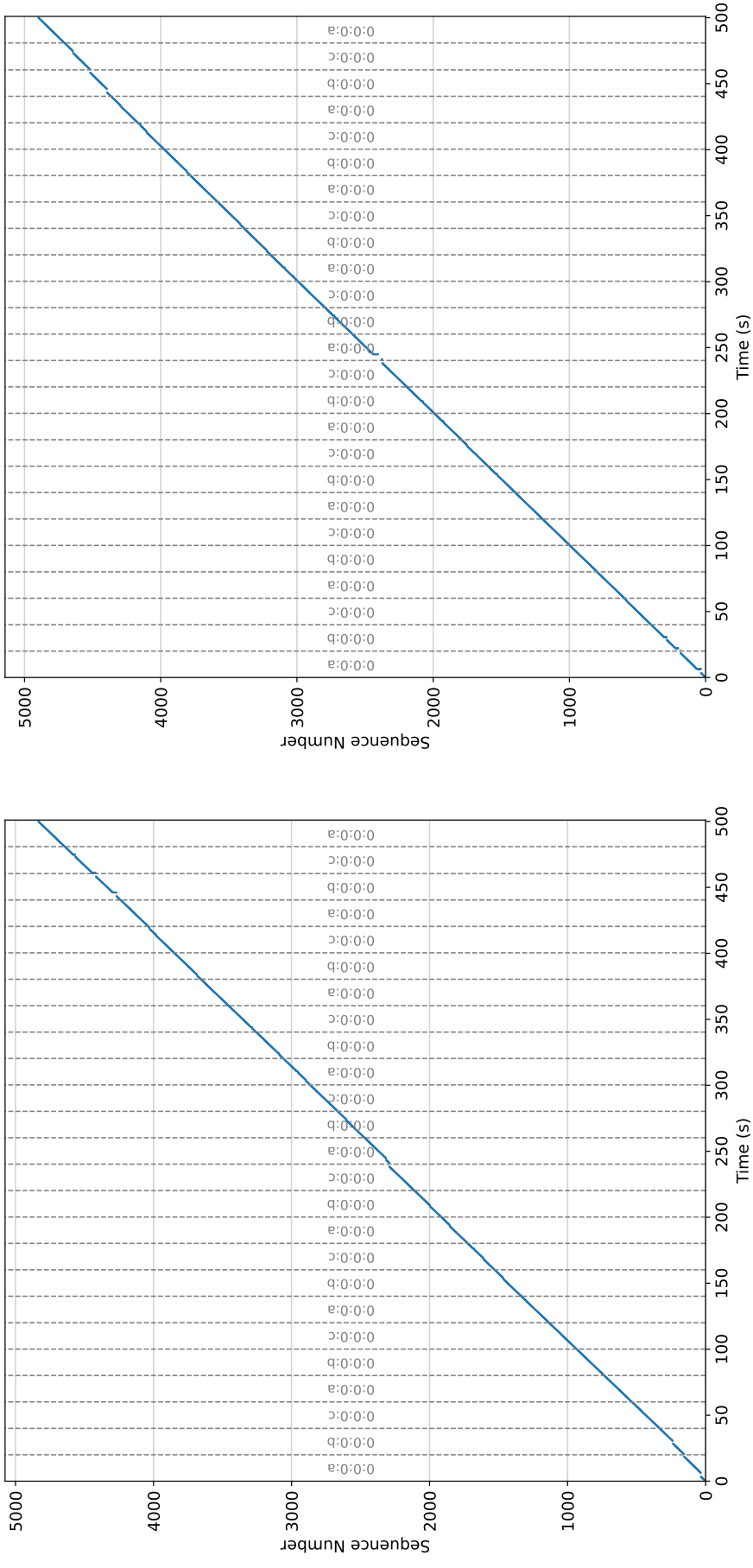
When receiving a packet that has a destination identifier matching the node this ignores it if it's from a locator the node is not currently connected to (`locs_joined` does not include locators connected to due to the soft handoff) and the destination locator does not match the interface on which it was received. This allows packets that are sent to the old locator to still be received - the whole point of the soft handoff. It will only stop duplicate packets being received from the old locator when they are destined for a different locator, i.e. the new locator.

A.6 Systems Issues

When running experiments, an issue with the system stability was encountered. Taking experiment 3 as an example, as described in section 5.2, figure A.2 shows how the received sequence numbers are mostly linear, but there are horizontal gaps in the sequence numbers received, and there are sometimes subsequent spikes, likely due to buffering on one of the nodes. There is no loss, however. Figure A.3 shows the throughputs, where these issues are a lot more noticeable. There are drops in throughput, corresponding to horizontal gaps in the graph, and sometimes subsequent spikes, corresponding to the spikes in received sequence numbers.

Note that figure A.3 has been trimmed to only show the interesting parts of the group, or else the spikes would make the pattern hard to see. Figure A.3b has 1 spike above 30kB/s, 1 spike above 40 kB/s, and 2 spikes above 50kB/s. Figure A.3a has 1 spike above 70kB/s, and 4 spikes above 40kB/s.

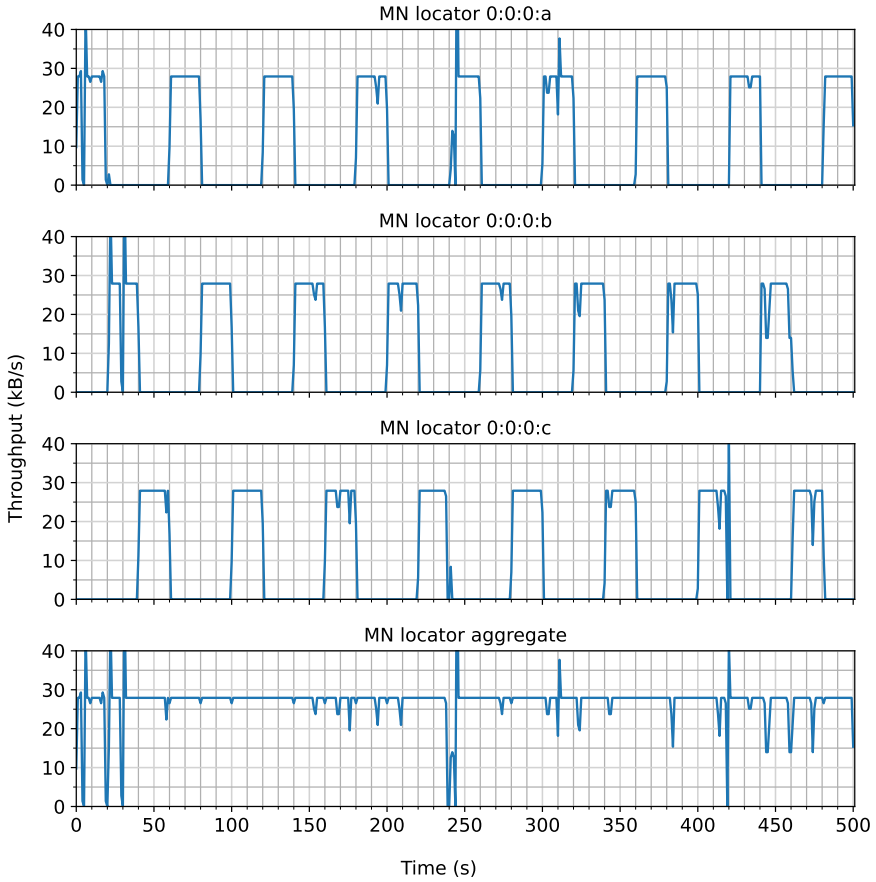
See the `system_issues_results` directory for the complete log files, graphs, and script used to create the graphs. Note this is a different `process.py` script than used for the included experimental results in section 5.2. This is because the implementation used to create these logs used a send buffer, so the application layer on the MN did not know what locator packets were sent on. This required cross-referencing sequence numbers with the source locator of packets received on the CN received. More details, and the reasoning behind later removing the send buffer, are discussed in section 4.2.



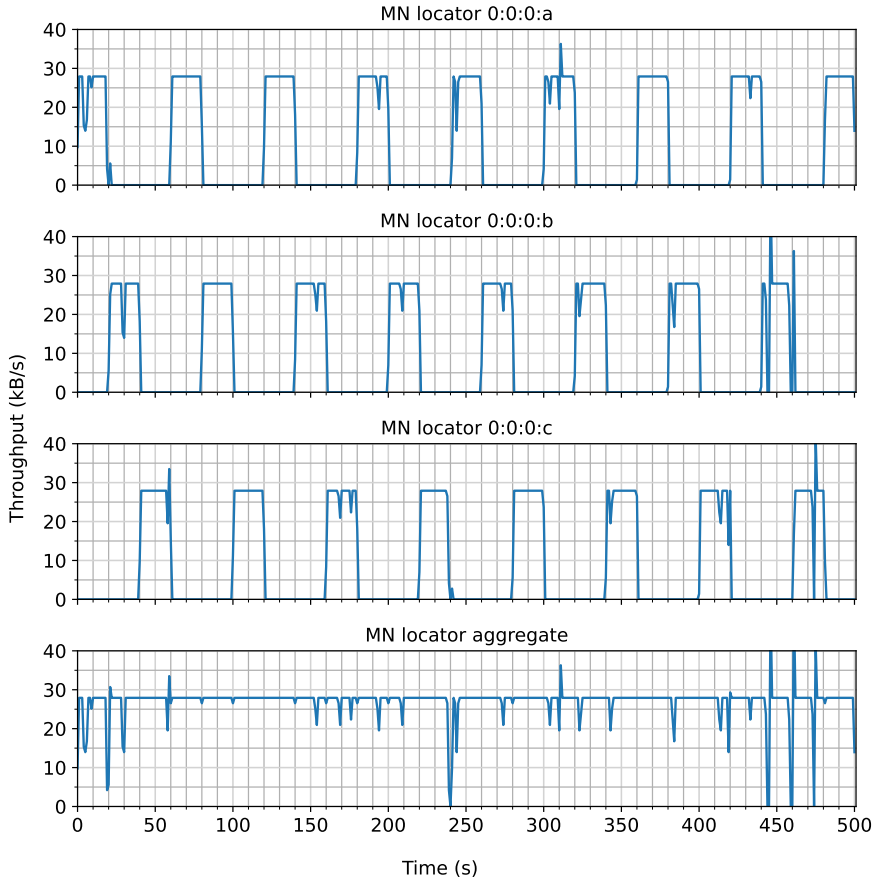
(a) Received sequence numbers vs Time on MN

(b) Received sequence numbers vs Time on CN

Figure A.2: System Issues Experiment 3 CN<->MN
Received sequence numbers vs Time



(a) Throughput in 1s buckets vs Time on CN



(b) Throughput in 1s buckets vs Time on MN

Figure A.3: System Issues Experiment 3 CN<->MN
Throughputs in 1s buckets vs Time

As the main focus of this project is obviously networking that was the first area assumed to be where the problem lay, as a scheduling or buffering issue. But the UDP send was not blocking, and the threading and thread synchronisation were working perfectly. The process was tried pinned to a specific CPU core with `$ taskset 0x1 <program>` to no avail. Using `tcpdump` showed the same gaps in packets sent and received on the CN, router, and MN.

Running `top` on the Pi while running showed that when systems issues occurred (printed as a warning by the experiment program) the process was in a 'D' state. This means it was in an uninterruptible sleep, due to I/O, otherwise data corruption could occur. As network issues were already ruled out, the only other I/O was logging. A long D state seems to be a common issue in Network File Systems (NFS), but that is not used here. A system request to display the list of blocked (D state) tasks with `echo w >/proc/sysrq-trigger` was made when the process was running. The relevant section of the kernel log from this is:

```
$ dmesg
...
[6367695.195711] sysrq: Show Blocked State
[6367695.199742]   task                        PC stack   pid father
[6367695.199791] jbd2/mmcblk0p2- D    0   824      2 0x00000028
[6367695.199801] Call trace:
[6367695.199818]  __switch_to+0x108/0x1c0
[6367695.199828]  __schedule+0x328/0x828
[6367695.199835]  schedule+0x4c/0xe8
[6367695.199843]  io_schedule+0x24/0x90
[6367695.199850]  bit_wait_io+0x20/0x60
[6367695.199857]  __wait_on_bit+0x80/0xf0
[6367695.199864]  out_of_line_wait_on_bit+0xa8/0xd8
[6367695.199872]  __wait_on_buffer+0x40/0x50
[6367695.199881]  jbd2_journal_commit_transaction+0xdf0/0x19f0
[6367695.199889]  kjournald2+0xc4/0x268
[6367695.199897]  kthread+0x150/0x170
[6367695.199904]  ret_from_fork+0x10/0x18
[6367695.199957] kworker/1:1     D    0 378944      2 0x00000028
[6367695.199984] Workqueue: events dbs_work_handler
[6367695.199990] Call trace:
[6367695.199998]  __switch_to+0x108/0x1c0
[6367695.200004]  __schedule+0x328/0x828
[6367695.200011]  schedule+0x4c/0xe8
[6367695.200019]  schedule_timeout+0x15c/0x368
[6367695.200026]  wait_for_completion_timeout+0xa0/0x120
[6367695.200034]  mbox_send_message+0xa8/0x120
[6367695.200042]  rpi_firmware_transaction+0x6c/0x110
[6367695.200048]  rpi_firmware_property_list+0xbc/0x178
[6367695.200055]  rpi_firmware_property+0x78/0x110
[6367695.200063]  raspberrypi_fw_set_rate+0x5c/0xd8
[6367695.200070]  clk_change_rate+0xdc/0x500
[6367695.200077]  clk_core_set_rate_nolock+0x1cc/0x1f0
[6367695.200084]  clk_set_rate+0x3c/0xc0
[6367695.200090]  dev_pm_opp_set_rate+0x3d4/0x520
[6367695.200096]  set_target+0x4c/0x90
[6367695.200103]  __cpufreq_driver_target+0x2c8/0x678
[6367695.200110]  od_dbs_update+0xc4/0x1a0
[6367695.200116]  dbs_work_handler+0x48/0x80
[6367695.200123]  process_one_work+0x1c4/0x460
[6367695.200129]  worker_thread+0x54/0x428
[6367695.200136]  kthread+0x150/0x170
[6367695.200142]  ret_from_fork+0x10/0x1
[6367695.200155] python3         D    0 379325 379321 0x00000000
[6367695.200163] Call trace:
[6367695.200170]  __switch_to+0x108/0x1c0
[6367695.200177]  __schedule+0x328/0x828
[6367695.200184]  schedule+0x4c/0xe8
[6367695.200190]  io_schedule+0x24/0x90
[6367695.200197]  bit_wait_io+0x20/0x60
[6367695.200204]  __wait_on_bit+0x80/0xf0
[6367695.200210]  out_of_line_wait_on_bit+0xa8/0xd8
```

```
[6367695.200217]  do_get_write_access+0x438/0x5e8
[6367695.200224]  jbd2_journal_get_write_access+0x6c/0xc0
[6367695.200233]  __ext4_journal_get_write_access+0x40/0xa8
[6367695.200241]  ext4_reserve_inode_write+0xa8/0xf8
[6367695.200248]  ext4_mark_inode_dirty+0x68/0x248
[6367695.200255]  ext4_dirty_inode+0x54/0x78
[6367695.200262]  __mark_inode_dirty+0x268/0x4a8
[6367695.200269]  generic_update_time+0xb0/0xf8
[6367695.200275]  file_update_time+0xf8/0x138
[6367695.200284]  __generic_file_write_iter+0x94/0x1e8
[6367695.200290]  ext4_file_write_iter+0xb4/0x338
[6367695.200298]  new_sync_write+0x104/0x1b0
[6367695.200305]  __vfs_write+0x78/0x90
[6367695.200312]  vfs_write+0xe8/0x1c8
[6367695.200318]  ksys_write+0x7c/0x108
[6367695.200324]  __arm64_sys_write+0x28/0x38
[6367695.200330]  el0_svc_common.constprop.0+0x84/0x218
[6367695.200336]  el0_svc_handler+0x38/0xa0
[6367695.200342]  el0_svc+0x10/0x2d4
```

Looking at the `python3` task stacktrace:

- `jbd2` is the thread that updates the filesystem journal, and `ext4` is the default Ubuntu file system (as well as a lot of other distributions)
- We can see than an inode is marked as dirty with `ext4_mark_inode_dirty`, and a file written with `ext4_file_write_iter`, and then a virtual file system write `vfs_write` is translated into an ARM write `__arm64_sys_write`.
  So this is happening during a file write.
- In ARM, `svc` means supervisor call, and `el0` exception level 0 (the lowest level of exception), so some sort of exception occurs and is then handled with `el0_svc_handler`.

Running `strace -r -t -v -p <PID of process>`, we can see the writes that take an exceptionally long amount of time. Here is an example where the write of 288 bytes to file descriptor 5 executes successfully but takes 2.24 seconds to complete:

```
21:47:28.684124 (+     0.000226) write(7, "2021-04-10 21:47:28.684061 [0:0:"..., 194) = 194
21:47:28.684381 (+     0.000256) write(1, "2021-04-10 21:47:28.684308 [alic"..., 122) = 122
21:47:28.684583 (+     0.000202) write(1, "\n", 1) = 1
21:47:28.684786 (+     0.000202) pselect6(0, NULL, NULL, NULL, {tv_sec=0, tv_nsec=5647000}, NULL) = 0 (Timeout)
21:47:28.690796 (+     0.006023) pselect6(0, NULL, NULL, NULL, {tv_sec=0, tv_nsec=0}, NULL) = 0 (Timeout)
21:47:30.930965 (+     2.240200) write(5, "2021-04-10 21:47:30.930813 0:0:0"..., 228) = 228
21:47:30.931427 (+     0.000433) getuid() = 1000
21:47:30.931812 (+     0.000385) socket(AF_UNIX, SOCK_DGRAM|SOCK_CLOEXEC, 0) = 9
21:47:30.932142 (+     0.000328) ioctl(9, SIOCGIFINDEX, {ifr_name="eth0", }) = 0
21:47:30.932506 (+     0.000364) close(9) = 0
21:47:30.933208 (+     0.000705) write(4, "2021-04-10 21:47:30.933090 [ff12"..., 348) = 348
```

So the problem seems to be exceptions that sometimes occur during file writes, which take a long time to resolve. These block the process executing by putting it in a D state until the write returns, affecting the system stability. These exceptions being the cause would make sense, as these issues aren't occurring consistently, but rather intermittently. This is happening on the MN, on the router, and on the CN; so its effect is being amplified 3 times. These exceptions are likely due to the page cache being flushed to disk, combined with poor performance of the Pi's SD cards. But finding the root cause would require more investigation. Regardless, we now know enough to fix the problem.

Removing the logging improved the system stability, but the issues still occurred with reduced frequency. This is because the experimental log is written to `stdout`, and `stdout` is piped to disk.

We were running our program on the Pi's through SSH piping `stdout` to a file, like this:

```
$ ssh <host> "<run> > <experiment_log_file>"
```

Charging this to:

```
$ ssh <host> "<run> | cat > <experiment_log_file>"
```

Fixed the issue once and for all. See `scripts/run_all.sh` and `scripts/run.sh` for details.

This essentially spawns another process to write to the file, and lets bash buffer between them. When an I/O exception occurs the writing process is put in a D state until the exception is handled, but the Python process is unaffected as its output is buffered until the writing process is able to read from it again.

B User Manual

B.1 Running

The only dependency of this program is Python3. Python version 3.8.5 was used for testing and experiments.

To run the heartbeat program, from the root directory of the project run:

```
$ python3 src/heartbeat.py <config file>
```

Similarly to run the experiment program, from the root directory of the project run:

```
$ python3 src/experiment.py <config file>
```

Note that this will require setting up the configuration files as described in section B.2. The `<config file>` command line parameter is an optional command line parameter to specify the configuration file. If it's not specified then `./config/config.ini` will be used, path relative to the root directory of the project.

B.2 Config files

The program is configured through `.ini` files. Here is an example:

```
[link]
log = true

# Local UDP multicast link layer emulation configuration
mcast_port      = 10000
mcast_interface = eth0
# MTU = 1440
buffer_size     = 1440

[network]
log = true

# Locators to join (which correspond to multicast
   ↪ addresses)
# Comma separated for joining multiple
# Hyphen separated for moving through
locators = 0:0:0:a,0:0:0:b,0:0:0:c

# Identifier of node
nid = ffff:0:0:a

# Optional, default value provided below
default_hop_limit = 3
```

```
# Time in seconds that backwards learning mappings will
   ↪ persist
# Note this is related to discovery.wait_time
# Optional, default value provided below
backwards_learning_ttl = 30

# Time in seconds that nodes will be considered
# to be in an active unicast session for after
# receiving or sending a packet to a node
# Used for sending locator updates
# Optional, default value provided below
active_unicast_session_ttl = 30

# Time between node moving locators in seconds
# Only used if there are hyphen separated sets of
   ↪ locators
# Optional, default value provided below
move_time = 20

# Soft handoff duration in seconds during which
# the node will be connected to both the old and new
   ↪ locators
# Only used if there are hyphen separated sets of
   ↪ locators
# Optional, default value provided below
handoff_time = 10

# Number of seconds to wait for a locator acknowledgement
# after sending a locator update
# Optional, default value provided below
loc_update_retry_wait_time = 1

# Number of times to retry sending a locator update
# Optional, default value provided below
loc_update_retries = 3

[transport]
log = true

[discovery]
log = true

hostname = alice
wait_time = 30

[application]
port = 1000
run_time = 510
```

Note:

- `link.mcast_interface` must match the interface on which the program is to

communicate with IP multicast.

- `link.mcast_port` is the UDP port used and must be the same on all running instances of the programs.

- `link.buffer_size` is the maximum size of packets sent via our emulated link layer. If this is larger than the MTU UDP may split our packets up into multiple IP packets resulting in undesirable behaviour like higher loss, as if any IP packet is lost the entire UDP packet is lost.

- `network.locators` specifies the locators the nodes should join. This is parsed by splitting on hyphens (`-`) which the node will cycle through based on `network.move_time` and `network.handoff_time`. If there are comma-separated locators the node will join all these locators.

  With `locators = 0:0:0:a,0:0:0:f-0:0:0:b-0:0:0:c` the node will:

  - at T=0s join `0:0:0:a` **and** `0:0:0:f`
  - at T=20s join `0:0:0:b`
  - at T=30s leave `0:0:0:a` **and** `0:0:0:b`
  - at T=40s join `0:0:0:c`
  - at T=50s leave `0:0:0:b`
  - at T=60s join `0:0:0:a` **and** `0:0:0:f`
  - at T=70s leave `0:0:0:c`

  The cycle will then repeat.

- `discovery.hostname` is the name of the host in the overlay network.

- `discovery.wait_time` determines the time between discovery messages and has a default value of 30 seconds.

- `application.port` port is the port used in the overlay network for the STP.

- `application.run_time` is used by `experiment.py` to terminate after the given number of seconds.

- The log flag for each layer determines if logs are taken at that layer. See section B.3.

B.3 Logging

Logs can be configured to be taken for each layer as shown in section B.2. Logs are saved to `logs/<hostname>_<layer>.log`.

The format of the log depends on the layer. See examples of logs in `heartbeat_logs`. The logs from the experiment were parsed to create the graphs shown in chapter 5.

B.4 Scripts

Numerous scripts were created to automate the testing and experimental processes. See the `scripts` directory for scripts for the Pis, and `scripts/desktop_scripts` for scripts using the workstation PC as a router. This includes deploying code to the Pis over with `rsync`, removing logs on the Pis, running processes, killing processes, and retrieving logs. These were run from a workstation and can use either Ethernet or Wifi.

B.5 Hardware Setup

The process for configuring the Pis was:

1. Flash SD card with Ubuntu Server LTS 20.04.2 On first login set password to `<PASSWORD>`
2. Change hostname with `hostnamectl set-hostname <NAME>`
3. Configure network
   (a) `echo "network: config: disabled" > /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg`
   (b) Assign static ethernet IP address and configure wifi connection with IPv6 disabled:
   `/etc/netplan/50-cloud-init.yaml`:

```
network:
    ethernets:
        eth0:
            dhcp4: false
            optional: true
            addresses: [<ETH_IPv6_ADDR>/64]
    wifis:
        wlan0:
                # disable IPv6
                link-local: []
                dhcp4: true
                optional: true
                access-points:
                    "<WIFI NETWORK>":
                        password: "<WIFI PASSWORD>"
    version: 2
```

   (c) `systemctl start avahi-daemon.service`
4. `sudo reboot`
5. `echo "<ETH_IPv6_ADDR> <NAME>" >> /etc/hosts` (on all machines)
6. Add `<NAME>` to list of hostnames in `~/.ssh/config` to config ssh user as ubuntu

7. `ssh-copy-id -i ~/.ssh/id_rsa.pub <NAME>`

On the workstation, `etc/hosts` contained:

```
fe80::dea6:32ff:fec4:67d5 alice-eth
fe80::dea6:32ff:fec4:6719 bob-eth
192.168.0.117 alice-wifi
192.168.0.118 bob-wifi
fe80::82ee:73ff:fe4a:393f base-station-eth
```

And `~/.ssh/config` contained:

```
# applies to mDNS hostname resolutions (e.g. alice)
# and manually configured ethernet connections (e.g. alice-eth)
Host alice* bob*
        User ubuntu
Host base-station*
        User root
Host base-station-luci-tunnel-eth
Hostname base-station-eth
LocalForward 127.0.0.1:8000 127.0.0.1:80
```

Note that base-station machine was unused for the experiments due to issues discussed in C.

B.6 Directories

- `src` contains the source code of the overlay network.
- `config` contains configaton files for instances of the program on the three Pis (Alice, Bob, and Clare), and the desktop workstation, for the three experiments and the heartbeat program.
- `scripts` contains scripts to run the heartbeat program or one of the experiments on the three Pis.

  `scripts/desktop_scripts` contains scripts to run them on two Pis (Alice and Bob) and the workstation PC with the PC acting as a router.
- `data_processing` is where these scripts retrieve the logs from the Pis to.
- `results` contains the final results of the experiments used in the report in the form of log files and graphs.

  `data_processing/process.py` contains the script used to create the graphs from these logs.
- `system_issues_results` contains logs, graphs, and the script used to create the graphs from the logs from when system stability issues effected the project.
- `heartbeat_logs` contains logs from a test run of the heartbeat program.

C Covid-19 Statement

This statement describes how the Covid-19 pandemic has affected this project.

It wasn't known at the start of the project that there wouldn't be access to lab space or space to set up a testbed in the labs. In fact, restrictions to building access actually increased in the second semester - when most of the experimental work was done.

For this reason, the networking hardware had to be set up in my bedroom. Saleem very kindly dropped off hardware at my flat in St Andrews, and I also picked some up from him on my bicycle. Flashing Ubuntu Server on the Pis, setting them up via a HDMI to USB webcam adapter, and connecting them via Ethernet provided to be relatively straightforward.

In contrast, WiFi proved more difficult than expected. Connecting the Pis to my landlord's WiFi router for a control plane connection and Internet access worked with some configuration. But issues arose when trying to set up an old shuttle PC from the labs running OpenWRT with a PCI USB expansion slot containing USB WiFi dongles as WiFi base stations.

As a result, instead of performing the experiments using WiFi, more relevant to the IoT context of the scenario, Ethernet was used instead.

The cost of this has been a less appropriate evaluative scenario and a large number of hours lost trying to configure WiFi base stations on OpenWRT.

References

[1] R. Atkinson and S. Bhatti. *ICMP Locator Update Message for the Identifier-Locator Network Protocol for IPv6 (ILNPv6)*. RFC 6743. RFC Editor, Nov. 2012. URL: https://www.rfc-editor.org/rfc/rfc6743.txt.

[2] R. Atkinson and S. Bhatti. *Identifier-Locator Network Protocol (ILNP) Architectural Description*. RFC 6740. RFC Editor, Nov. 2012. URL: https://www.rfc-editor.org/rfc/rfc6740.txt.

[3] R. Atkinson and S. Bhatti. *Identifier-Locator Network Protocol (ILNP) Engineering Considerations*. RFC 6741. RFC Editor, Nov. 2012. URL: https://www.rfc-editor.org/rfc/rfc6741.txt.

[4] R. Atkinson and S. Bhatti. *IPv4 Options for the Identifier-Locator Network Protocol (ILNP)*. RFC 1958. RFC Editor, Nov. 2012. URL: https://www.rfc-editor.org/rfc/rfc6746.txt.

[5] R. Atkinson and S. Bhatti. *IPv6 Nonce Destination Option for the Identifier-Locator Network Protocol for IPv6 (ILNPv6)*. RFC 6744. RFC Editor, Nov. 2012. URL: https://www.rfc-editor.org/rfc/rfc6744.txt.

[6] R. Atkinson and S. Bhatti. *Optional Advanced Deployment Scenarios for the Identifier-Locator Network Protocol (ILNP)*. RFC 1958. RFC Editor, Nov. 2012. URL: https://www.rfc-editor.org/rfc/rfc6748.txt.

[7] R. Atkinson, S. Bhatti, and S. Rose. *DNS Resource Records for the Identifier-Locator Network Protocol (ILNP)*. RFC 6742. RFC Editor, Nov. 2012. URL: https://www.rfc-editor.org/rfc/rfc6742.txt.

[8] S. N. Bhatti, R. Atkinson, and J. Klemets. "Integrating Challenged Networks". In: *MILCOM*. 2011.

[9] S. N. Bhatti and R. Yanagida. "Seamless Internet connectivity for ubiquitous communication". In: *PURBA UBICOMP*. 2019.

[10] B. Carpenter. *Architectural Principles of the Internet*. RFC 1958. RFC Editor, June 1996. URL: https://www.rfc-editor.org/rfc/rfc1958.txt.

[11] C. Curino et al. "TinyLIME: bridging mobile and sensor networks through middleware". In: *Third IEEE International Conference on Pervasive Computing and Communications*. 2005, pp. 61–72. DOI: 10.1109/PERCOM.2005.48.

[12] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 1958. RFC Editor, July 2017. URL: https://www.rfc-editor.org/rfc/rfc8200.txt.

[13] A. Dunkels, B. Gronvall, and T. Voigt. "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors". In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. LCN '04. USA: IEEE Computer Society, 2004, pp. 455–462. ISBN: 0769522602. DOI: 10.1109/LCN.2004.38. URL: https://doi.org/10.1109/LCN.2004.38.

[14] *IBM 1981 Product fact sheet*. https://www.ibm.com/ibm/history/exhibits/pc25/pc25_fact.html. Accessed: 11/04/2021.

[15] P. Levis et al. "TinyOS: An Operating System for Sensor Networks". In: vol. 00. Jan. 2005, pp. 115–148. ISBN: 978-3-540-23867-6. DOI: 10.1007/3-540-27139-2_7.

[16] T. Li. *Recommendation for a Routing Architecture*. RFC 1958. RFC Editor, Feb. 2011. URL: https://www.rfc-editor.org/rfc/rfc6115.txt.

[17] T. Narten et al. *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. RFC Editor, Sept. 2007. URL: https://www.rfc-editor.org/rfc/rfc4861.txt.

[18] D. Phoomikiattisak and S. Bhatti. "IP-layer soft handoff implementation in ILNP". In: *MobiArch 2014 - Proceedings of the 9th ACM MobiCom Workshop on Mobility in the Evolving Internet Architecture* (Sept. 2014). DOI: 10.1145/2645892.2645895.

[19] J. Postel. *Internet Protocol*. RFC 1958. RFC Editor, Sept. 1981. URL: https://www.rfc-editor.org/rfc/rfc791.txt.

[20] *Raspberry Pi FAQs*. https://www.raspberrypi.org/documentation/faqs/. Accessed: 11/04/2021.

[21] M. Weiser. "Some Computer Science Issues in Ubiquitous Computing". In: *Communications of the ACM* (1993).