Implementing Emulated Communication Models for
Hybrid and Dynamic Network Topologies

by

Joseph Murphy

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in

Electrical and Computer Engineering
by

_____

December 2022

APPROVED:

_____
Professor Alexander Wyglinski, Research Advisor

_____
Professor Carlo Pinciroli

_____
Professor Bashima Islam

Abstract

In this thesis, network emulation is presented as a solution to testing, developing, and extending communication systems in a time- and cost-effective manner. Complex hybrid and dynamic wireless networks require extensive testing that is not easily conducted in hardware testbeds and may not be modeled accurately enough in network simulation tools. Network emulation provides the benefits of both hardware testbeds and simulation tools while also minimizing the shortcomings of each. This thesis evaluates the Extendable Mobile Ad-hoc Network Emulator (EMANE) as a network emulation tool by assessing its ability to emulate several complex network models. These models include hybrid wireless rural broadband deployments, an intelligent routing software development environment, and dynamic robot swarm networks. The emulated models were determined to be accurate enough to their hardware counterparts such that EMANE can be used as an effective tool for prototyping and testing communication systems.

Acknowledgements

I would first and foremost like to thank Professor Alexander Wyglinski for all the advice and guidance he has provided me with, not only throughout this thesis and graduate degree, but also my entire senior year and capstone design project. Without his insights and advice, this project would not be complete.

I would like to thank Professor Carlo Pinciroli and Professor Bashima Islam for serving as my research committee members and providing me with feedback on my work.

Thank you to both the teams at US Ignite and the U.S. Army DEVCOM for supporting my graduate degree and providing me with the tools necessary to complete my research.

And lastly, a special thank you to all my friends and family for their endless support throughout my entire academic career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The need for wireless communications and network technology is rapidly growing. As more and more devices become network-enabled, the need for technology to support this rapid growth in wireless communication systems becomes apparent. Despite this need for interconnectivity, there is still a large divide Networks are expensive to deploy and test, especially in rural environments where hybrid combinations of wireless technologies must be used and when using special network topologies like dynamic mobile ad-hoc networks. This creates a need for a low-cost, easy method to do initial testing on networks and network technologies to validate viability before spending money on hardware deployments and testing Use network simulation and emulation for preliminary testing as it can require little to no hardware, can be conducted in a lab, and costs less than building physical networks for each new experiment [2–4].

1.2 State of the Art

Many software and combination software-hardware platforms exist for testing networks. ns-3, GNURadio, OMNeT++, CORE

1.3   Current Issues

Why are these simulators not as good?

- Many are not free or open-source (expensive to use and possibly not as customizable)

- Can be complex to set up

- Often only focus on network layer and abstract MAC/PHY layer, OR model the MAC/PHY layer but does not allow for integration with network software and protocols

1.4   Thesis Contribution

- The Extendable Mobile Ad-hoc Network Emulator (EMANE) is proposed as a valuable testing tool that addresses issues with other similar networking simulation tools. An overview of installing and using the tool is provided.

- An initial program designed to maximize bandwidth usage in a constrained wireless network is developed. It is shown how EMANE can be used as a network software development environment.

- Basic integration between EMANE and the robot swarm simulator ARGoS is developed. EMANE is shown to be capable of extending and enhancing other tools to provide accurate communication emulation.

1.5   Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the network emulator EMANE and the motivation behind the selection of this tool. An overview of the how to use EMANE and its subsystems is presented. Three different use-cases for the EMANE tool are considered to evaluate the effectiveness of the tool. Chapter 3 proposes the first use case for testing with EMANE, testing rural broadband deployments. Two similar network topologies are proposed and tested with the help of EMANE. Chapter

4 explores a second use case for utilizing EMANE, development of networking technologies and systems. In this case a program for more intelligent allocated limited bandwidth is developed. Chapter 5 finally details a third use case for EMANE, integrating with other simulation tools to provide more accurate communication models. The paper is concluded with a summary of work completed and recommendations for future work in Chapter 6.

Chapter 2

Overview on Network Emulation

Before utilizing the EMANE tool and presenting several situations the tool can be used for, we must first be understood why EMANE was selected and why network emulation is used over simulation or hardware testbeds in this thesis. After justifying the choices behind selecting EMANE, an in-depth tutorial on the installation, configuration, and operation of the tool is presented. Finally, a brief overview of mobile ad-hoc network (MANET) routing protocols is presented. These protocols are essential to understand as they are commonly used in EMANE to create routes between nodes.

2.1   Testing Communication Networks

There are typically three ways new network architectures, technologies, and protocols are developed and tested. These are network simulation, network emulation, and hardware testbeds [5]. As expected each of the three methods has pros and cons.

Hardware testbeds are the most accurate since they encompass the devices expected to be used in the network once development and testing is done. Hardware, however, is expensive to purchase, time-consuming to deploy, and often difficult to troubleshoot if errors do not consistently appear [6]. This makes hardware testing not accessible to users that have a low budget.

Network simulation is one solution to testing that appears to solve many of the issues with testing on hardware. Several free and open-source network simulators like ns-3 [7] or

OMNeT++ [8] are commonly use and provide a solution to the high costs of hardware. Like most network simulators, these simulators operate on the concept that the behavior of a network and its components can be modeled via statistical and mathematical models. Creating models for network behavior allows simulators to run faster than real time since the models do not need to wait for effects to actually happen. The caveat to this, however, is that simulation models need to be highly accurate when developed or else results from the simulation will not match expected hardware behavior. Researchers creating new simulation models must ensure the models are validated against the expected hardware behavior to ensure the accuracy of the models, before they are can be used in testing [9]. Simulation also has the benefit of being highly repeatable since the behavior of the network can be more tightly controlled and any random processes can be set up to repeat previous random outputs [5].

Network emulation exists somewhere between testing on hardware and testing inside a simulation. These emulators are still software that gets used to mirror the behavior of a testbed like simulators, but emulators operate on real network data instead of modeling the behavior of a network. Because emulation testbeds operate on actual network traffic, they also have the ability to interface with hardware allowing hardware testing without building a full hardware network. This characteristic of operating on real network traffic also has the downside of introducing more computation overhead. The system emulating the network needs to not only manage all the test traffic, but also manage all the effects and permutations that are imparted on the traffic.

Table 2.1 summarizes the pros and cons of different testing environments.

2.2   Evaluation of Network Testing Tools

There are several network simulation and emulation tools that were considered for use in this thesis during initial research. These programs all provide functionality that would achieve the goals of performing inexpensive and less time-consuming network testing, but were eventually decided against in favor of EMANE. This section will highlight a few of these tools and explain the reasoning behind why they were not selected, before finishing

Table 2.1: Pros and Cons of Different Types of Network Testing

Testbed Type	Pros	Cons
Hardware	- Highly accurate - Does not require modification of networking software	- Expensive to build - Time consuming to deploy and configure - Errors can be sporadic
Simulation	- Free tools available - Can run faster than real time - Easy to reconfigure and modify	- Models must be designed to be highly accurate - Software must be translated to a simulation model
Emulation	- Free tools available - Can run native implementations of network software - Can interface with hardware	- Must run in real time - Requires higher computational overhead

by introducing EMANE and the driving reasons for its selection.

One of the most common tools, ns-3, is a discrete event-based network simulation tool that is commonly used for simulation of TCP/IP networks [7]. It is the successor to the ns-2 tool [10], with the two major differences being the method that simulation scripts are written and how these scripts are executed in the simulator. In ns-3 the core simulator as well as experiment scripts are written in C++ and ns-3 provides bindings for Python so simpler scripts can also be written in Python. This is much simpler than ns-2 which required the use of Tcl scripts to create experiments [11]. The migration from ns-2's method of programming to ns-3's makes the tool easier to use, but it does still require an amount of programming knowledge to understand how utilize the library of models provided. Being one of the most commonly used tools, ns-3's library of simulation modules have been extensively validated, but these modules typically focus on the network layer and above. ns-3 does have the ability to be used as an emulator, however the motivation emulation mode in ns-3 was primarily driven by the ability to use ns-3 with hardware [12]. Since we would like to avoid hardware wireless equipment, this option for ns-3 is not valuable for the research present in this paper. The wide-scale use and extensive support of ns-3 made the tool a promising candidate, but it was decided against due to the abstractions made at the physical layer.

OMNeT++ is another discrete event simulator written in C++, like ns-3, however it is not specifically designed as a communications network simulator [8]. Despite not being designed as a communications network simulator, the ability to create plugins for the tool

has led to many researchers creating models of communication networks that can be used for simulation. The issue with this collection of plugins is that many users have found them to be highly incompatible with each other. This is likely due to the isolated nature many of the models were developed under [13] One of the advantages of OMNeT++ is that is supports a graphical user interface, that allows for easier building of and interaction with network testbeds. Figure 2.1 shows an example of this GUI, which is based off of the Eclipse IDE.
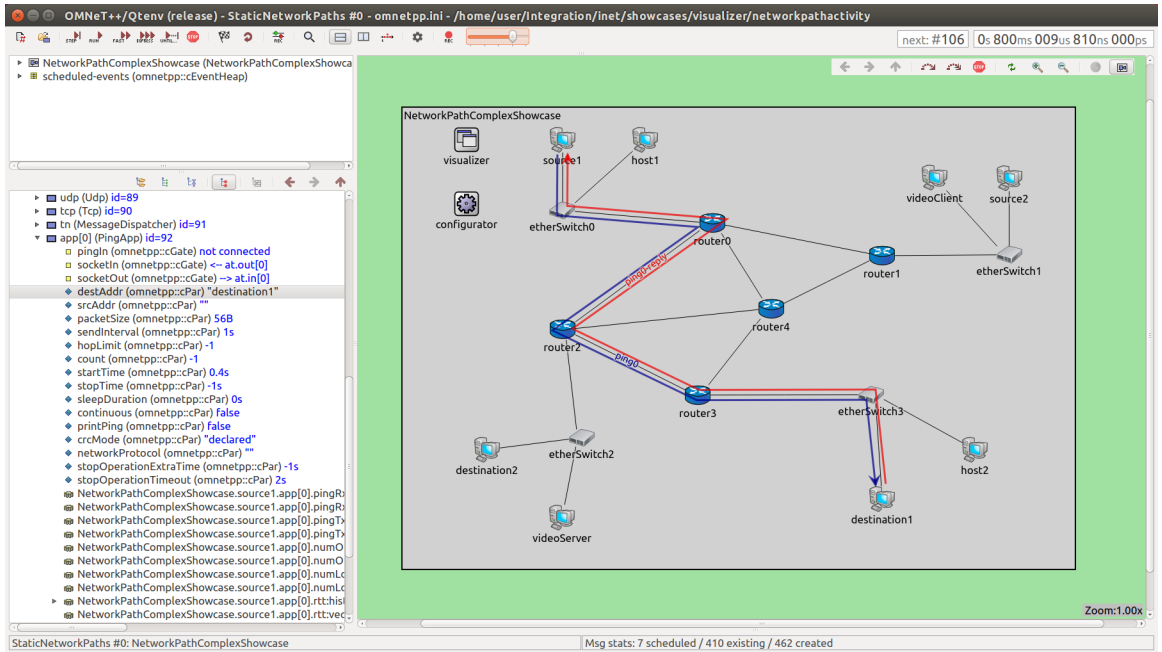
Figure 2.1: An example of the OMNeT++ Simulator GUI.

The Common Open Research Emulator (CORE) is an emulation tool focused on emulation of layers three (network layer) and above in the OSI stack [14]. Of all the tools presented so far, CORE is one of the easiest to use. Similar to OMNeT++, CORE provides the user with a GUI that takes the form of a blank canvas where users can drag and drop preconfigured nodes such as routers and servers. Figure 2.2 is a basic example of what a typical CORE session looks like. CORE also gives users the ability to create virtual nodes via a Python framework, that is useful for more complex scenarios. Without any modification, CORE emulates the network layer and above perfectly, as each virtual node is running

the actual, unmodified software that would be running on hardware [1]. Where CORE runs into issues is when wireless channels are introduced. By default, CORE operates on the concept that nodes that are close enough to communicate they have a connection, and if they are far enough by a certain distance, they can not communicate. This simplification is not acceptable for a majority of testing that requires accurate wireless communication modeling, and so to solve this issue EMANE was integrated into CORE so that all physical and data link layer emulation happened through EMANE [1]. Figure 2.3 shows the configuration menu for EMANE, within CORE. With this pairing CORE and EMANE become a very valuable tool, however in this thesis CORE is not used, independently or alongside EMANE. The main reasoning for this is because a majority of the protocols being used at the network layer or higher, are not present in CORE and computational complexity can be saved by running them in EMANE directly, without CORE.

EMANE is a network emulation tool originally developed by the Naval Research Lab and currently maintained by AdjacentLink LLC [15]. The tool is a discrete event-driven emulator programmed in C++ and Python, and is configured by the user primarily through XML files and Bash scripts. The software was developed with the intention of creating a platform that could emulate the physical and data link layers of the OSI network model with high accuracy, avoiding many of the abstractions made by other tools. EMANE consists of several subsystems and components required to create a fully functional testbed, and this complexity can lead to an initial steep learning curve with the tool. The tool is open-source, however, the online community around EMANE is rather small and most of the discussion and troubleshooting surrounding the tool is only found on EMANE's GitHub issues page, not helping with the complexity issue. Despite all this, once the user forms a core understanding of the tools and systems within the software, the tool can be used to effectively and quickly create model wireless networks. EMANE's ability to be configured through XML makes deployment of networks very rapid, and because the included models are pluggable, these configuration files can be reused completely independently of the topology of the nodes in the network. For these reasons and the reasons highlighted previously regarding other testing tools, EMANE was selected as the tool of choice for this thesis and the next section is dedicated to understanding how to both install the tool, and how all the
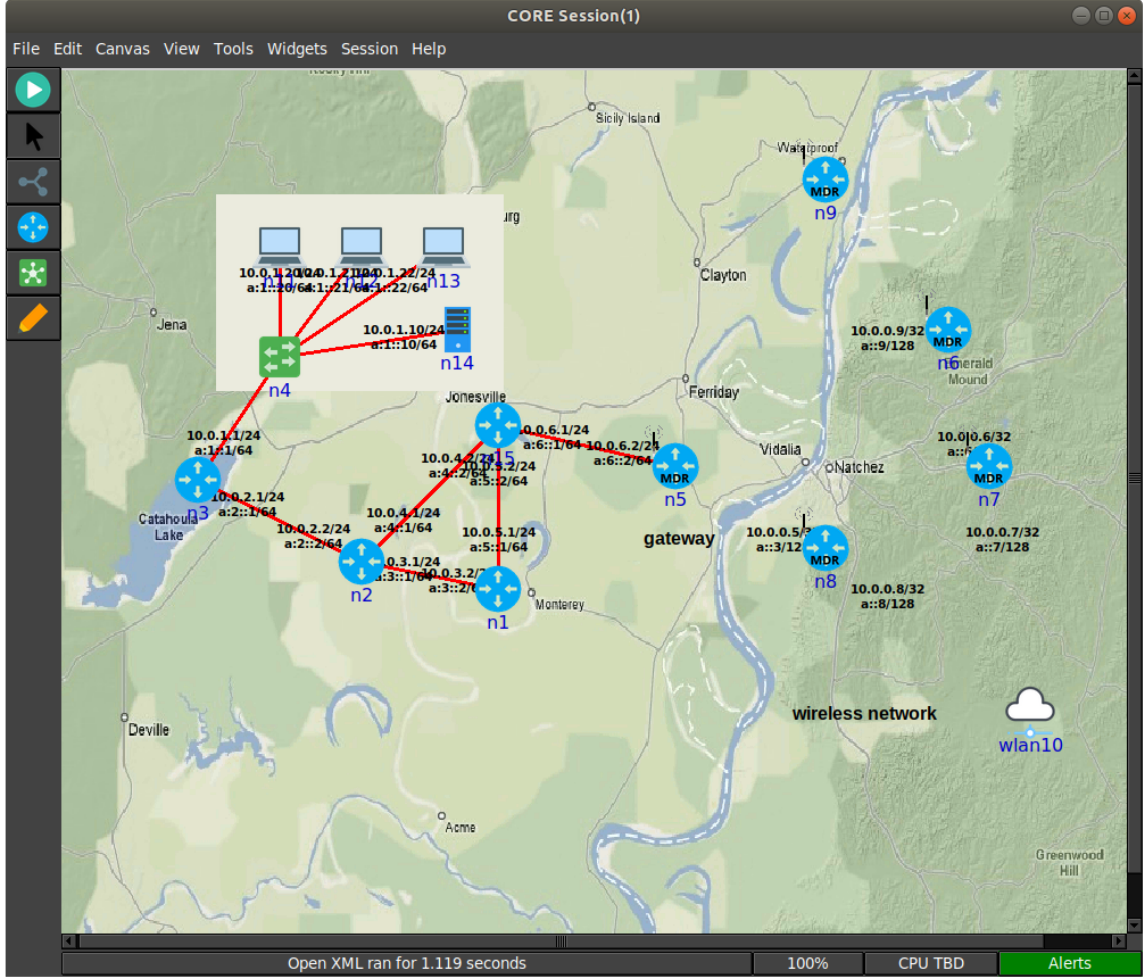
Figure 2.2: An example of the CORE Emulator GUI.

pieces work together.

Table 2.2 highlights the main advantages and disadvantages of each of these tools.

2.3 Using EMANE

Having selected EMANE for use in this thesis, we must now understand how to install, configure, and operate the tool. This section will begin by detailing the installation of EMANE, EMANE can be installed and used in a few distinct ways. The primary two methods are installation via the bundle of pre-built binaries provided by AdjacentLink
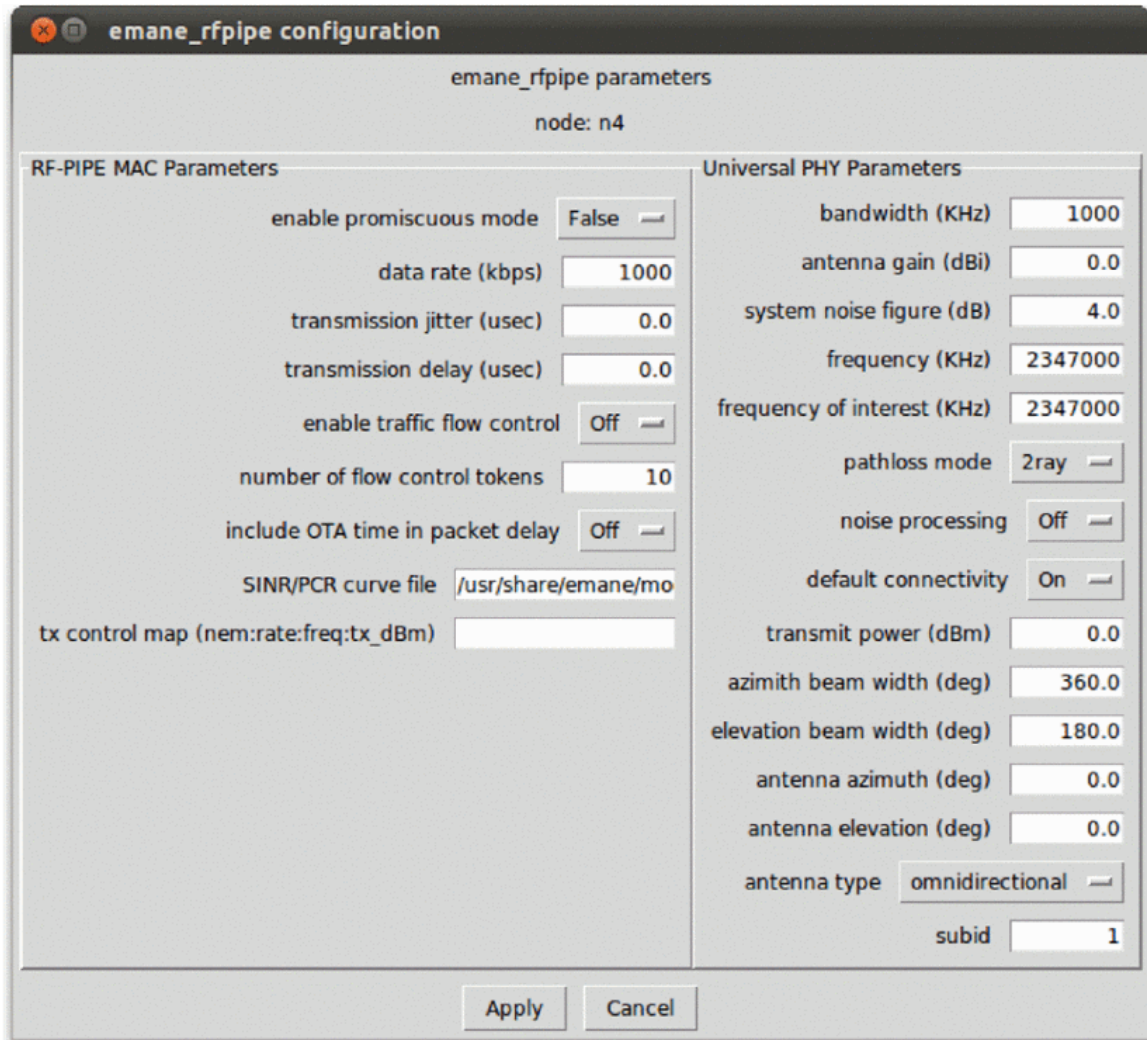
Figure 2.3: The configuration menu for EMANE, within CORE. From [1].

or building all the required programs from source. Compiling the software from source is typically only necessary when making extensions to the tool, such as adding custom modules. Since only the default included modules are used, the precompiled bundle is sufficient for the work completed in this thesis. The installation instructions for EMANE can be found in [16], but it should be noted that these instructions are sometimes out of date. The full EMANE GitHub repository [17] may also provide guidance that is more up to date. EMANE version 1.3.3 was the primary version of the tool used in this work, and supports the Rocky 8, Fedora 37, and Ubuntu 20.04 Linux distributions.

Table 2.2: Overview of Advantages and Disadvantages of Different Networking Testing Tools

Tool	Advantages	Disadvantages
ns-3	Open-source Extensive protocol library Widely used and validated	Abstracts the PHY and MAC layers Limited control of individual nodes Simple wireless propagation models
OMNeT++	GUI-based Large library of models	Not purpose built for network simulation Provided models are often not compatible
CORE	Open-source GUI-based Simple models do not require programming knowledge	Only emulates the network layer and above Limited network protocols available by default
EMANE	Open-source Purpose built to emulate PHY and MAC layer Can use any implementation of network software	Small online community Initial steep learning curve Requires extensive computing resources for large-scale networks

These instructions assume a fresh installation of Ubuntu 20.04.04 is being used and that is system is up-to-date with the latest software. If a different supported distribution of Linux is used, the steps will be similar, however the commands will differ. Refer to [16,17] for further details. See Appendix A for a list of the specific commands run to complete the entire installation process

1. The first step is downloading the precompiled binaries. As seen in Figure 2.4, the *wget* utility is used to download the compressed file, but any method for retrieving this file can be used. This compressed file should be extracted before the next step.

2. To install the downloaded binaries, the *dpkg* application is used. Errors will be reported upon installation, as not all the dependencies are installed. This will be fixed after the initial installation via *apt*. Figure 2.5 shows the initial installation finishing with errors, and the command used to fix these errors.

3. Lastly, we can verify that EMANE was installed correctly by having the program output its version. Figure 2.6 shows that EMANE version 1.3.3 was successfully installed.

4. Once installation of EMANE has been verified, additional support software (like testing tools and routing protocols) can be installed to use with EMANE. Appendix A

shows some examples of installing certain tools.

Figure 2.4: Downloading the precompiled EMANE binaries using the *wget* command.

Figure 2.5: Installing the EMANE program *dpkg* command.

Now that EMANE is installed, it is important to understand the major systems that work together to enable network emulation. The main structure in EMANE that everything operates around is known as the Network Emulation Module (NEM). Each NEM can be thought of as a single network node, similar to a singular radio. As each NEM is an independent network node, every NEM created requires network stack isolation within the kernel of the host system. All the traffic flowing through the emulation testbed consists of

Figure 2.6: Verifying EMANE installed correctly by displaying the version number.

real IP packets and are therefore treated like regular network traffic by the kernel. If the network stacks were not isolated, packets would not route through the emulator and would instead just be switched between processes in the kernel, bypassing all wireless channel effects.

There are several methods than can be used to create network stack isolation. Full virtual machines could be used, but these are very computationally inefficient and would not allow the emulated testbed to maintain a large amount of nodes. Additionally, nodes in EMANE do not need the full isolation provided by virtual machines, and it is even beneficial if the file system could be shared. To solve this problem, containers are used as the primary method of isolation. [18] examines several types of virtualization that can be used for isolation nodes in CORE, but the same concepts apply to EMANE. Between FreeBSD jails, Linux OpenVZ containers, and Linux namespaces containers, the namespaces containers are found to be the most efficient. For most examples of EMANE and all the testbeds created in this thesis, Linux Containers (LXCs) are used. These are lightweight containers that are built on top of Linux namespaces and allow for the sharing of files and other resources, while keeping processes and the network stack isolated.

Figure 2.7 shows what a typical EMANE emulation node will look like, with one of these structures existing per LXC. In this diagram, the NEM is visible, with all of its surrounding subsystems and connections. The blue boxes within the NEM are the emulation models that are responsible for imparting wireless channel effects on packets moving upstream and downstream through the emulator. The green boxes represent the transport boundary. This is the edge of the emulation node where packets leave the emulator and return to normal application space. The orange box represents the event service that is responsible

for changing the state and settings of the emulator during runtime in order to create effects in the network. These three systems are the major subsystems that make up EMANE and allow it to create highly dynamic networks, used to test a variety of use cases and will be examined in depth in the next three subsections.
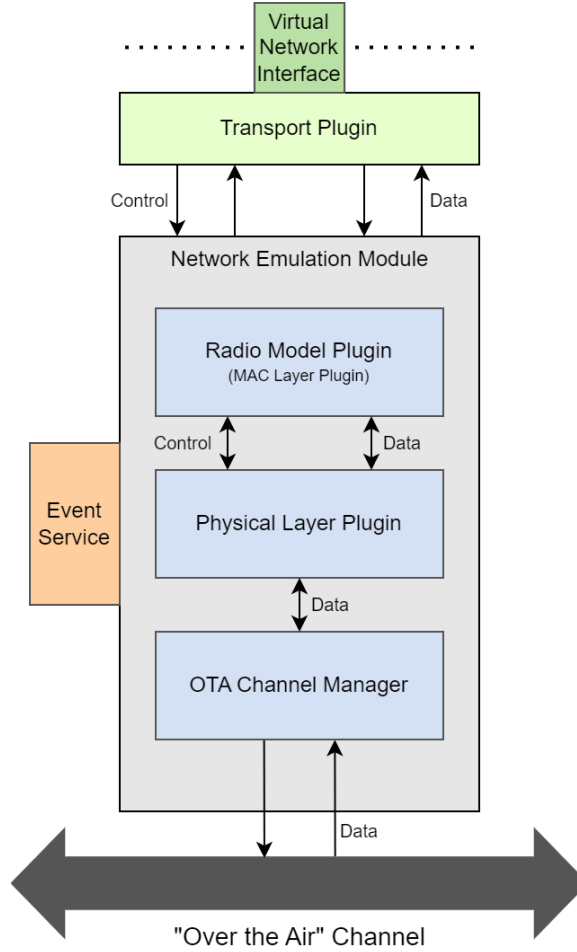
Figure 2.7: An overview of an individual EMANE emulated network node.

2.3.1 Emulation Model Processing

The emulation model processing is the system primarily responsible for modifying packets traveling through the NEM to mimic the behavior of a wireless signal. This system of EMANE consists of two main parts, the Physical Layer Plugin and the Radio Model Plugin, both of which can be seen in blue in Figure 2.7. As the names of the plugins imply, the

Physical Layer plugin is responsible for imparting the effects of the physical layer, and the Radio model plugin is responsible for the MAC Layer. The Over the Air Channel Manager is primarily just responsible for pulling packets addressed to the NEM, off of the shared bus, and putting packets ready to be transmitted onto the shared channel. This shared OTA channel simply takes the form of a virtual interface bridge that all the NEMs are attached to, creating a bus topology. The NEMs use a multicast scheme to listen for packets as other control channels also use this interface bridge to communicate. It is the main backbone of the emulation that connects all the individual network nodes, both for control messages, but also data payloads.

As previously mentioned, the shared PHY plugin is responsible for the effects of the physical layer. It is possible to create a custom Physical Layer model, but is usually not necessary and is the only PHY model included in EMANE. This model is responsible for attributes like pathloss and signal propagation, fading, noise modeling, the antenna profile.

The first parameter to be set is the propagation model. This model is responsible for calculating the expected pathloss between two nodes, and has three options. The first two options are *freespace* and *2ray*. These use location data contained within the node and the freespace or 2-ray flat earth model to calculate the pathloss of a channel. The third option is called *precomputed* and is used when the pathloss is to be calculated external to the emulator. This is useful if a more complex model is to be used, or a different tool is being used to model signal propagation.

The second configuration step is related to the power characteristics of the node and its virtual antenna. Transmit power can be set, and serves the same purpose it would on a hardware radio. The antenna gains can be set as a static value, or a more complex profile that contains a list of antenna pattern entries. The following is an example of the contents of an antenna profile file from the CORE/EMANE documentation [14]:

```
<!-- 30degree sector antenna pattern with main beam at +6dB and gain
↪   decreasing by 3dB every 5 degrees in elevation or bearing.-->
<antennaprofile>
  <antennapattern>
    <elevation min='-90' max='-16'>
      <bearing min='0' max='359'>
        <gain value='-200'/>
      </bearing>
    </elevation>
    <elevation min='-15' max='-11'>
      <bearing min='0' max='5'>
        <gain value='0'/>
      </bearing>
      <bearing min='6' max='10'>
        <gain value='-3'/>
      </bearing>
```

The final piece of the PHY model that must be understood is noise modeling. EMANE models noise by taking the transmission power of any NEM transmitting, and adding it to the noise floor of the receiving node if the receiver's frequency is within the bandwidth of the interfering signal. This results in all interfering signals being treated as white noise [19]. EMANE provides parameters to set if the noise mode is for all signals within the correct frequency, just signals that are out-of-band, or turning off noise processing completely. Once all of these factors are set, EMANE can then use the following two equations to determine if a packet can actually be received. If the received power ($rxPower$) is greater than the receiver's sensitivity ($rxSensitivity$), the packet is received.

$$rxPower = txPower + txAntennaGain + rxAntennaGain - pathloss$$

$$rxSensitivity = -174 + noiseFigure + 10log(bandwidth)$$

Figure 2.8 shows what a general Physical Layer Plugin configuration file will look like.

The second piece of the emulation model processing layer of EMANE is the Radio Model plugins. Unlike the PHY plugin which is shared for all NEMs, the Radio plugin is different

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE phy SYSTEM 'file:///usr/share/emane/dtd/phy.dtd'>
<phy>
    <param name="bandwidth"                      value="1000000"/>
    <param name="excludesamesubidfromfilterenable"  value="on"/>
    <param name="fading.model"                   value="none"/>
    <param name="fading.nakagami.distance0"      value="100.000000"/>
    <param name="fading.nakagami.distance1"      value="250.000000"/>
    <param name="fading.nakagami.m0"             value="0.750000"/>
    <param name="fading.nakagami.m1"             value="1.000000"/>
    <param name="fading.nakagami.m2"             value="200.000000"/>
    <param name="fixedantennagain"               value="38.000000"/>
    <param name="fixedantennagainenable"         value="on"/>
    <param name="frequency"                      value="2347000000"/>
    <param name="frequencyofinterest"            value="2347000000"/>
    <param name="noisebinsize"                   value="20"/>
    <param name="noisemaxclampenable"            value="off"/>
    <param name="noisemaxmessagepropagation"     value="200000"/>
    <param name="noisemaxsegmentduration"        value="1000000"/>
    <param name="noisemaxsegmentoffset"          value="300000"/>
    <param name="noisemode"                      value="none"/>
    <param name="propagationmodel"               value="precomputed"/>
    <param name="subid"                          value="1"/>
    <param name="systemnoisefigure"              value="4.000000"/>
    <param name="timesyncthreshold"              value="10000"/>
    <param name="txpower"                        value="0.000000"/>
</phy>
```

Figure 2.8: A generic configuration file for an EMANE PHY Plugin

depending on the type of waveform emulation desired. EMANE ships with four Radio plugins:

- rfPipe - A generic wireless channel that does not do channel access functions
- IEEE802.11abg - A model specifically for 2.4GHz Wi-Fi waveforms
- TDMA - A generic time division multiple access scheme
- Bypass - A model specifically for testing that passes traffic along to the next layer unchanged

Other plugins can be created by extending the emulator, and a plugin designed for LTE and 5G is currently under development in collaboration with the srsRAN project.

These four plugins all operate differently, but for the purposes of this thesis only the rfPipe model will be used. The goal of the rfPipe model is to create a simple model that handles data rate, delay, jitter, and probability of packet loss due to signal to interference and noise ratio (SINR). The data rate, delay, and jitter are all simple parameters that get set and are implemented by holding packets at the radio model layer long enough to achieve the set values. The Packet Completion Rate (PCR) utilizes the signal, interference, and noise power levels calculated by the Physical Layer model to find the SINR value, and compares that to a lookup table. The portion of the table and corresponding curves, as seen in Figure 2.9, are used to come up with a probability value that is used to decided if the packet should be lost or otherwise corrupted due to noise.
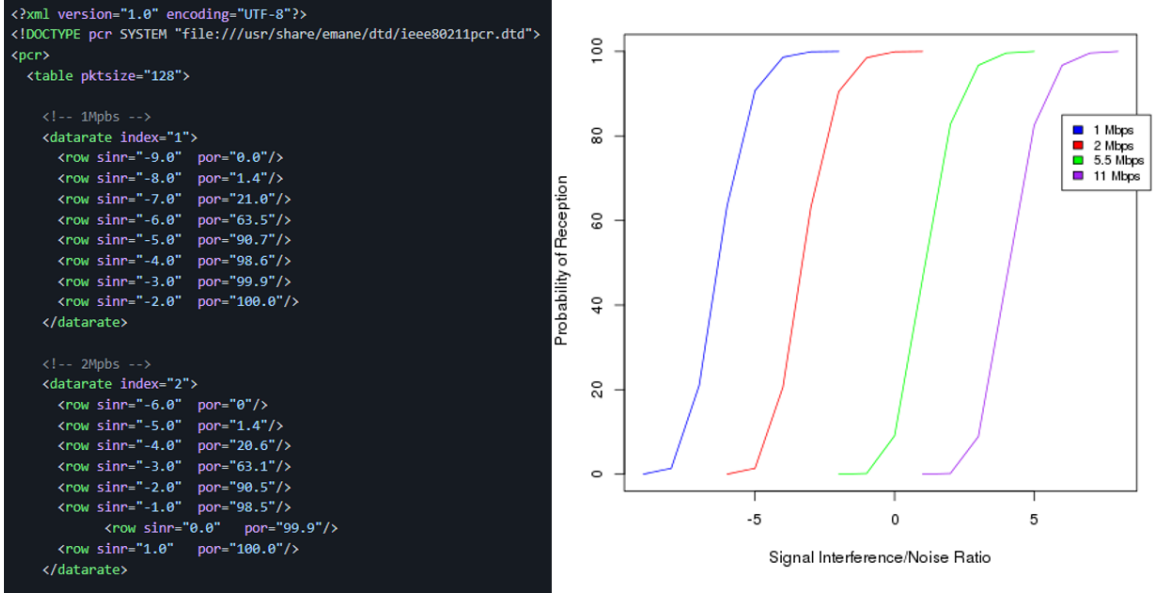
Figure 2.9: The packet complete rate (PCR) table and corresponding curve.

One of the essential pieces of the emulation processing system is ensuring the models used are accurate to hardware, to ensure results used from the emulator are accurate. While several studies use the rfPipe model [20–22], very little literature could be found validating the models included within EMANE. The rfPipe model is generic enough that there are no modulation schemes or other access functions that need to be validated, but the implementation of the behavior must still be checked. Similarly, the calculations performed

at the Physical Layer are fairly simple, but must also be validated. A validation like the one found for the TDMA model [23], should be performed for all models presently in EMANE, as well as for any custom models created. For the purposes of this thesis, EMANE is being used to model parameters like throughput and latency which can be easily compared to hardware equivalent testbeds. Chapter 3 will examine this to make comparisons between the two, however, if EMANE is to be used in more advanced testbeds with highly complex models, validation is essential.

2.3.2   Transport Boundary Processing

The second system required to operate EMANE is the transport boundary. The transport boundary is represented in Figure 2.7 by the green boxes, and is responsible for handing packets entering and exiting the emulator. Since EMANE is capable on operating on actual TCP/IP packets, one of the main roles of the transport boundary is to translate TCP/IP packets entering the emulator into something EMANE can operate on, and translate EMANE's packet back into TCP/IP packets. There are two types of transports that can be used with EMANE, raw transports and virtual transports.

Raw transports are used for enabling the hardware in the loop functionality of EMANE. When a raw transport plugin is initialized, it is connected to a hardware network interface present on the emulation host machine. This breaks the convention of having EMANE nodes exist within LXCs, but if only one EMANE node is present on the host the node can run external to a virtualized environment. If multiple nodes exist, a pair of virtual network interfaces can be created and bridged to the hardware interface to create a tunnel from inside the LXC to the hardware interface on the host. This method is used in Chapters 3 and 4 and will be outlined in more detail there. Figure 2.10 shows an example of the layout of the network interface on the host and the EMANE instance.

The second type of transport are virtual transport plugins. These are more commonly used as they are the plugins that carry from the emulator instance to application space via a virtual network interface. When EMANE is started on an LXC using a virtual transport, the LXC is created with a pair of virtual interfaces. One of the interfaces is internal to the LXC and is used as the endpoint for EMANE, the other is external to the LXC and
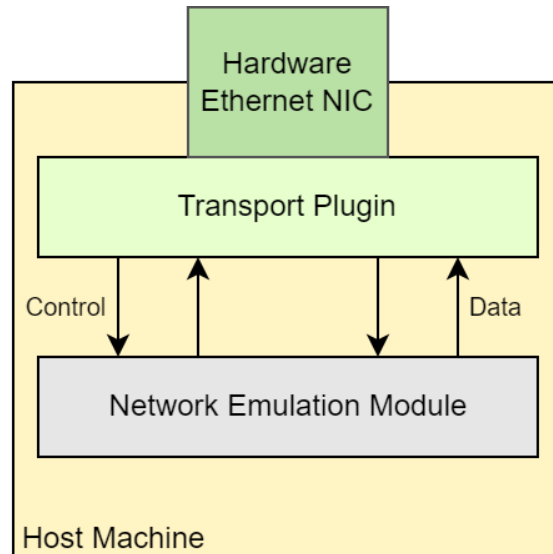
Figure 2.10: An example EMANE topology using the raw transport plugin.

is bridged together with all the other nodes to allow traffic to pass. Figure 2.11 shows an example of the layout of the virtual network interfaces bridged together between EMANE instances, internal on the host machine.
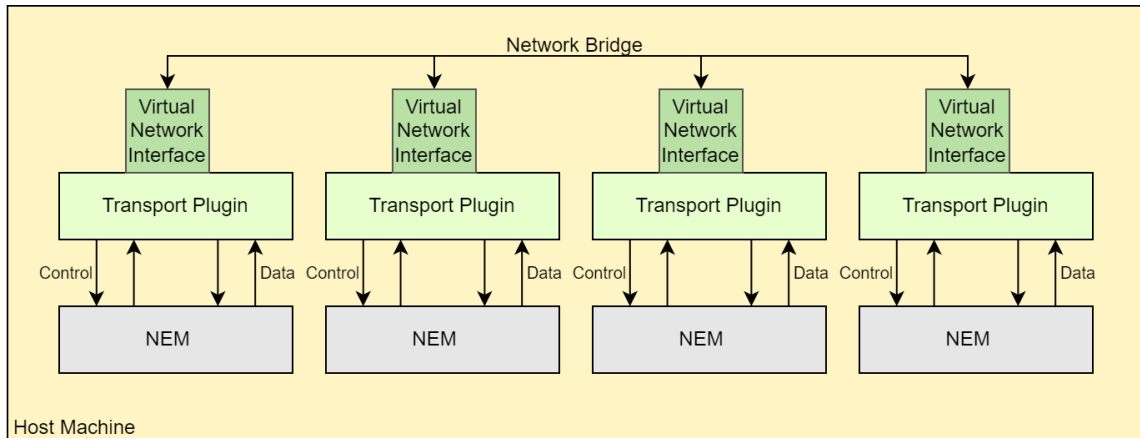

Figure 2.11: An example EMANE topology using the virtual transport plugin.

2.3.3   Event Processing

Event processing is the system in EMANE that allows parameters of the simulator to change during runtime. There are five main event types. The first type, pathloss, is used when the propagation model is set to *precomputed* and allows an external tool to make pathloss calculations. The second type, location, is used to move nodes around and primarily influences pathloss values that are calculated internally to the tool when using *freespace* or *2ray* settings. The location event can take the latitude, longitude, altitude, pitch, yaw, roll, and velocity data for a node. The third event type is the antenna profile event. Similar to the antenna profile setting in the Physical Layer plugin, this event can be used to feed antenna data to a node, and change that data during runtime. The fourth event is used to change the fading model being used. This event is fairly simple and is used to toggle whether a node uses the Nakagami fading model or no fading model. The final event is called Comm Effect, and is used to change generic communication parameters of a node, such as the latency, jitter, or probability of packet loss and duplication.

There are two primary methods that events get generated for distribution to nodes. The first is using an EEL file. This file is a specific EMANE file that is taken in at the start of emulation and contains "sentences" that outline event parameters and what time during the simulation they should begin. Figure 2.12 shows an example of an EEL file and the sentences contained within. The first value is the time the event is scheduled to fire, in this case all of these events fire at the start of emulation. The second value is the ID of the NEM the event is destined for, and this is followed by the event name and any corresponding parameters. The other main method for generating events is through Python bindings that allow direct subscription to the event channel for publishing of events. Events are passed to NEMs via the event channel, but all this channel is, is a different multicast service that the NEMs subscribe to on the same network interface bridge that the over-the-air channel lives on. By putting both of these channels on the same bridge, the complexity of managing several network devices per NEM is lessened.

```
0.0   nem:70 pathLoss nem:22,96.3 nem:23,95.0 nem:24,95.1 nem:25,95.2 nem:26,95.3 nem:27,95.4 nem:28,95.5 nem:29,95.0 n
0.0   nem:70 pathLoss nem:42,95.3 nem:43,95.4 nem:44,95.5 nem:45,95.0 nem:46,95.1 nem:47,95.2 nem:48,95.3 nem:49,95.4 n
0.0   nem:70 pathLoss nem:62,95.2 nem:63,95.3 nem:64,95.4 nem:65,94.2 nem:66,94.2 nem:67,96.3 nem:68,96.3 nem:69,123.3
0.0   nem:1 location gps 40.031075,-74.523518,3.000000
0.0   nem:2 location gps 40.031165,-74.523412,3.000000
0.0   nem:3 location gps 40.031227,-74.523247,3.000000
0.0   nem:4 location gps 40.031290,-74.523095,3.000000
0.0   nem:1 antennaprofile 1,0,90
0.0   nem:2 antennaprofile 1,0,270
0.0   nem:3 antennaprofile 1,0,90
0.0   nem:4 antennaprofile 1,0,270
0.0   nem:1 commeffect nem:2,1.050000,2.000300,10.000000,12.000000,45,54 nem:3,0.050000,0.025000,0.000000,0.000000,0,0
0.0   nem:2 commeffect nem:1,11.055000,22.000330,11.000000,13.000000,46,55 nem:3,0.000000,0.000000,0.000000,0.000000,10
0.0   nem:3 commeffect nem:1,0.050000,0.025000,0.000000,0.000000,0,0 nem:2,0.000000,0.000000,0.000000,0.000000,10000,10
0.0   nem:4 commeffect nem:1,0.000000,0.000000,50.000000,0.000000,0,0 nem:2,0.000000,0.000000,0.000000,0.000000,5000,50
0.0   nem:1 fadingselection nem:2,none nem:3,nakagami nem:4,none
```

Figure 2.12: An example of an EEL file provided to the EMANE event service.

2.4   Routing in Mobile Mesh Networks

EMANE was designed primarily to work with mobile ad-hoc networks, also known as MANETs, and while emulating other network types is possible, MANET routing protocols work well with EMANE's architecture. This special classification of network is characterized by its dynamic topology that often rapidly changes due to the mobility of network nodes and the tendency for the wireless links to connect and disconnect [24]. This lack of a fixed topology means that any node that exists in the network must be able to communicate without help from centralized infrastructure or a gateway and therefore must be able to independently make routing decisions. Since the topology of a MANET is a mesh, the primary method for traffic traveling through the network is through relaying. Each node in the mesh acts as a router and upon receiving network traffic, must determine if the traffic is destined for itself, or a different node in the network. In the second case, the relaying node will use its knowledge of the routing table and network topology to determine which neighbor node the packets must be forwarded to [25]. These types of routing protocols can be separated into two categories, proactive protocols and reactive protocols [25].

2.4.1   Proactive Mesh Routing

The first category of MANET routing protocol is the proactive protocol. Proactive protocols are similar to traditional routing protocols in the sense that they create and

maintain a routing table. By maintaining a routing table, any transmission that needs to be sent can be done so immediately since the most efficient route is known. This allows proactive protocols to operate with less latency than reactive MANET routing protocols as they do not need to wait for route discovery at the time of transmission [26]. The caveat to this is that these protocols require much higher control traffic as they must perform periodic link sensing to ensure the routing tables are up-to-date. In a network where bandwidth is constrained, it is essential to understand this limitation at the time of design as MANETs are often already bandwidth restricted and adding another high usage system could be too much.

There are two routing protocols that the maintainers of EMANE often use in examples and tutorials for the tool. These are the Open Link State Routing (OLSR) protocol [27] and the Better Approach to Mobile Ad-hoc Networking (B.A.T.M.A.N.) protocol [28]. These routing protocols are both proactive MANET routing protocols and are two of the more common purpose built proactive protocols found in MANETs. They both operate on the similar principle of link sensing via a discovery packet (called HELLO packets in OLSR and OGM packets in B.A.T.M.A.N.), but differ in how the best routes are calculated.

When deciding between these two protocols it is found that B.A.T.M.A.N. typically will outperform OLSR [29]. This can be attributed to the manner in which OLSR performs link sensing. When evaluating two paths, the path with the least number of relays is considered the best path in OLSR [27]. This concept is feasible in high speed wired networks where queuing and relaying of data is often the slowest portion of a packets journey, but in wireless networks where the quality of the medium can drastically vary, this does not work as well. B.A.T.M.A.N. attempts to solve this issue by using the concept that the link from which a discovery packet arrives from first, must be the best link to send a packet for it to arrive at the destination in the OGM. Since the link with the lowest latency and highest throughput is expected to deliver packets the fastest, it can be assumed it is the best link [28]. Since B.A.T.M.A.N. nodes only measure which neighbor has a route to a given destination, and does not share the topology of the entire network graph, the protocol also produces less control traffic, which contributes to it performing better. Figure 2.13 shows what topology information nodes might have in a network running B.A.T.M.A.N. Nodes B and C maintain

a list that indicates which nodes in the graph are accessible through each neighbor. In this example, node B does not know the layout of connections between nodes C, D, E, and F. It is only aware that C has the best route to all of those destinations, and relies on C's routing table to relay the packets the rest of the way. This has the benefit of making topological changes that occur on the opposite side of the mesh transparent to nodes not immediately effected, reducing the amount of traffic that needs to occur when the mesh changes.
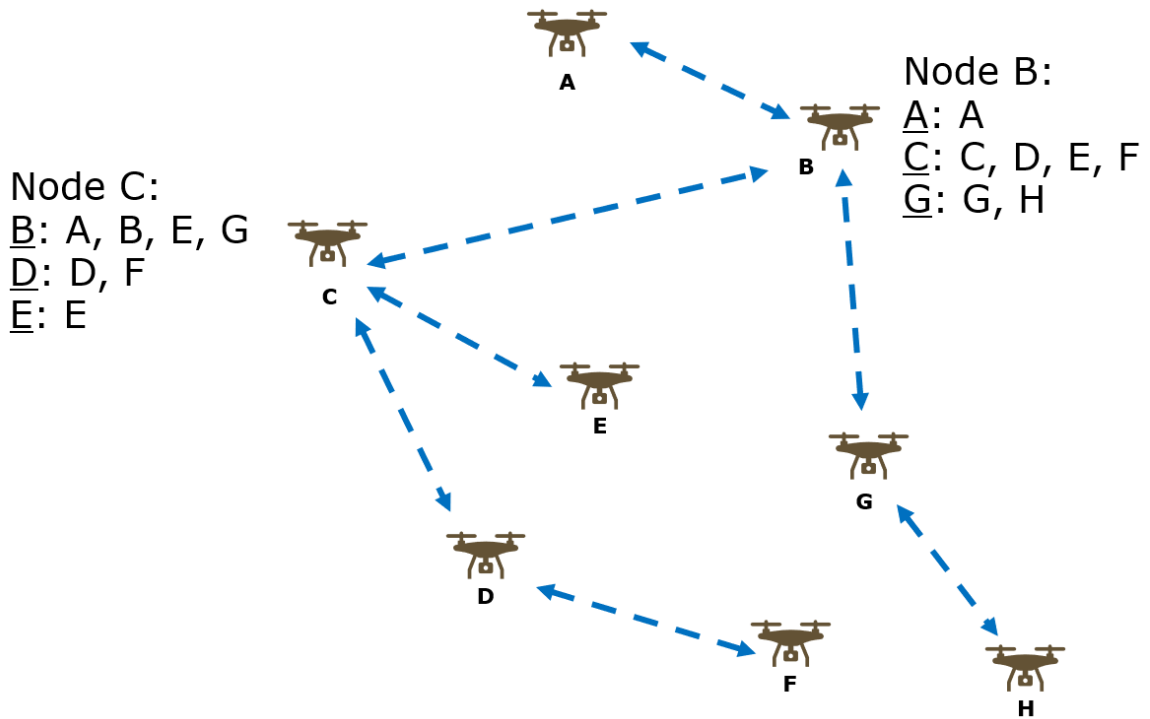
Figure 2.13: An overview of an individual EMANE emulated network node.

2.4.2 Reactive Routing

The either primary category of MANET routing protocol are reactive protocols, also sometimes referred to as "on-demand" protocols [25]. These protocols are referred to as on-demand protocols because routes to a destination are only found, when the transmitting node requires them. This has the benefit of greatly reducing control traffic present in the network since topology and link state information is not periodically shared, but it does result in higher latency as traffic must wait for a route to be found. This type of routing

protocol could be beneficial in a network that is constrained on bandwidth and will typically only contain traffic that is not time-sensitive.

Two examples of reactive MANET protocols are AODV and DSR [30]. These protocols generally act by sending out request packets upon needing a route, indicating the destination the traffic is intended for, and waiting for a response. Nodes that have a route to that destination response, and once a full route is found the traffic is sent along it. The route is then stored and considered a good route for that traffic until an attempt at using the route fails. At that point the discovery process is repeated finding a new route.

Generally, reactive protocols are found to not be as effective in highly mobile MANETs [30, 31]. B.A.T.M.A.N. is found to have less maximum available bandwidth in some situations when compared to AODV, but will more reliably deliver packets successfully, and often is able to react to changes in the graph faster. For this reason, proactive protocols like OLSR and B.A.T.M.A.N. are more commonly used in simple EMANE testbeds, as seen in the tutorial [16].

All the testbeds in the remainder of this thesis will use the batman-adv implementation of the B.A.T.M.A.N. protocol. This implementation is built into the Linux kernel and installation instructions for using it can be found in Appendix A.

2.5   Chapter Summary

This chapter covered necessary background information on the differences between testing networks in hardware, testing networks in simulation, and testing networks in emulation. Network hardware testbeds are expensive and so simulation or emulation were decided to be used instead. Several network simulators including ns-3, OMNeT++, CORE, and EMANE were examined, and the emulator EMANE was determined to be the best tool for this thesis thanks to its accurate modeling of the PHY and MAC layers, mechanisms to individual control the software running on each node, and ability to interface with hardware. Having selected EMANE, installation instructions were detailed, and an overview of the tool and its subsystems was presented. The chapter then finished by presenting an overview of MANET routing protocols, the type of protocol that will be used in the EMANE testbeds detailed

over the next three chapters.

Chapter 3

Hybrid Wireless Rural Broadband Networks

The first testing scenario EMANE was used for was emulating two wireless, hybrid network topologies that were designed to more effectively deliver broadband to small rural communities. The important factors to emulate

3.1 What was implemented?

3.1.1 OVERCOME Testbed

3.1.2 ZoomTEL Testbed

3.2 How was it implemented?

3.3 Why was it implemented this way?

3.4 Compare EMANE results to real testbed?

3.5 Chapter Summary

Table 3.1: mmWave Model EMANE Parameters

EMANE Parameter	Value
Delay	0.5ms
Data Rate	1Gbps
Frequency	60GHz
Channel Width	500MHz
Fixed Antenna Gain	43dBi
Pathloss Model	Freespace

Table 3.2: Ubiquiti LTU Model EMANE Parameters

EMANE Parameter	Value
Delay	2.5ms
Data Rate	200Mbps
Frequency	5.8GHz
Channel Width	20MHz
Fixed Antenna Gain	19.5dBi
Pathloss Model	Freespace

Chapter 4

Networking Software Development Environment

4.1 Intelligent Method of Bandwidth Distribution

4.2 Implementation Methodology

4.3 Why was it implemented this way?

4.4 Effectiveness of the Program

4.5 Chapter Summary

Chapter 5

Dynamic Robot Swarm Networks

5.1   Extending Existing Software

5.2   Integrating the Software

5.3   Integration Design Decisions

5.4   Integration Results

5.5   Chapter Summary

Chapter 6

Conclusion

6.1   Research Outcomes

6.2   Future Work

Bibliography

[1] J. Ahrenholz, T. Goff, and B. Adamson, "Integration of the core and emane network emulators," in *2011 - MILCOM 2011 Military Communications Conference*, 2011, pp. 1870–1875.

[2] "Internet/broadband fact sheet," 2021. [Online]. Available: https://www.pewresearch.org/internet/fact-sheet/internet-broadband/

[3] E. A. Vogels, "Some digital divides persist between rural, urban and suburban america," 2021. [Online]. Available: https://pewrsr.ch/3k5aU6J

[4] M. Shafiq, P. Singh, I. Ashraf, M. Ahmad, A. Ali, A. Irshad, M. Khalil Afzal, and J.-G. Choi, "Ranked sense multiple access control protocol for multichannel cognitive radio-based iot networks," *Sensors*, vol. 19, no. 7, 2019.

[5] S. Guruprasad, R. Ricci, and J. Lepreau, "Integrated network experimentation using simulation and emulation," in *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, 2005, pp. 204–212.

[6] M. Neufeld, A. Jain, and D. Grunwald, "Nsclick: Bridging network simulation and deployment," in *Proceedings of the 5th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWiM '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 74–81. [Online]. Available: https://doi.org/10.1145/570758.570772

[7] nsnam, "What is ns-3?" 2022. [Online]. Available: https://www.nsnam.org/about/

[8] OpenSim Ltd., "What is omnet++?" 2019. [Online]. Available: https://omnetpp.org/intro/

[9] P. A. B. Bautista, L. F. Urquiza-Aguiar, L. L. Cárdenas, and M. A. Igartua, "Large-scale simulations manager tool for omnet++: Expediting simulations and post-processing analysis," *IEEE Access*, vol. 8, pp. 159 291–159 306, 2020.

[10] J. Heidemann and T. Henderson, "The network simulator - ns-2," 2011. [Online]. Available: http://nsnam.sourceforge.net/wiki/index.php/User_Information.

[11] N. Kamoltham, K. N. Nakorn, and K. Rojviboonchai, "From ns-2 to ns-3 - implementation and evaluation," in *2012 Computing, Communications and Applications Conference*, 2012, pp. 35–40.

[12] S. Yadav, M. Gaur, and V. Laxmi, "Ns-3 emulation on orbit testbed," in *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2013, pp. 616–619.

[13] S. Gupta, M. Ghonge, D. Thakare, and P. Jawandhiya, "Open-source network simulation tools an overview," *International Journal of Advanced Research in Computer Engineering & Technology*, vol. 2, 04 2013.

[14] J. Ahrenholz and T. Goff, "Common open research emulator," 2022. [Online]. Available: https://coreemu.github.io/core/

[15] E. Schreiber, P. Kehoe, and S. Galgano, "Extendable mobile ad-hoc network emulator (emane)," 2018. [Online]. Available: https://www.nrl.navy.mil/Our-Work/Areas-of-Research/Information-Technology/NCS/EMANE

[16] "Emane-tutorial," 2018. [Online]. Available: https://github.com/adjacentlink/emane-tutorial

[17] "Emane," 2022. [Online]. Available: https://github.com/adjacentlink/emane

[18] J. Ahrenholz, "Comparison of core network emulation platforms," in *2010 - MILCOM 2010 MILITARY COMMUNICATIONS CONFERENCE*, 2010, pp. 166–171.

[19] T. Schucker, T. Bose, and B. Ryu, "Emulating wireless networks with high fidelity rf interference modeling," in *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, 2018, pp. 822–828.

[20] C. Kam, S. Kompella, G. D. Nguyen, J. E. Wieselthier, and A. Ephremides, "Modeling the age of information in emulated ad hoc networks," in *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, 2017, pp. 436–441.

[21] Y. E. Sagduyu, Y. Shi, T. Erpek, S. Soltani, S. J. Mackey, D. H. Cansever, M. P. Patel, B. F. Panettieri, B. K. Szymanski, and G. Cao, "Multilayer manet routing with social-cognitive learning," in *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, 2017, pp. 103–108.

[22] C. Kam, S. Kompella, and A. Ephremides, "Experimental evaluation of the age of information via emulation," in *MILCOM 2015 - 2015 IEEE Military Communications Conference*, 2015, pp. 1070–1075.

[23] A. Nikodemski, J.-F. Wagen, F. Buntschu, C. Gisler, and G. Bovet, "Reproducing measured manet radio performances using the emane framework," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 151–155, 2018.

[24] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The broadcast storm problem in a mobile ad hoc network." in *5th annual ACM/IEEE international conference on Mobile computing and networking*, vol. 8, 08 1999, pp. 151–162.

[25] S. Puri and V. Arora, "Routing protocols in manet: A survey," *International Journal of Computer Applications*, vol. 96, no. 13, 2014.

[26] S. Kaur and K. Arora, "Performance evaluation of diverse manet routing protocols-a review," *International Journal of Computer Applications*, vol. 107, no. 17, 2014.

[27] T. H. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," RFC 3626, October 2003. [Online]. Available: https://www.rfc-editor.org/info/rfc3626

[28] M. Lindner, S. Eckelmann, S. Wunderlich, M. Hundebøll, A. Quartulli, and L. Lüssing, "The b.a.t.m.a.n. project." [Online]. Available: http://www.open-mesh.org/

[29] M. Abolhasan, B. Hagelstein, and J. C.-P. Wang, "Real-world performance of current proactive multi-hop mesh protocols," in *2009 15th Asia-Pacific Conference on Communications*, 2009, pp. 44–47.

[30] D. Kaur and N. Kumar, "Comparative analysis of aodv, olsr, tora, dsr and dsdv routing protocols in mobile ad-hoc networks," *International Journal of Computer Network and Information Security*, vol. 5, no. 3, pp. 39–46, 03 2013.

[31] D. Seither, A. König, and M. Hollick, "Routing performance of wireless mesh networks: A practical evaluation of batman advanced," in *2011 IEEE 36th Conference on Local Computer Networks*, 2011, pp. 897–904.

Appendix A

Installation of EMANE

Appendix B

Intelligent Router Source Code

B.1 Control Script

```python
#!/usr/local/bin/python3.8
import time
import datetime
import logging

import classify
import allocate

if __name__ == '__main__':

    date = datetime.date.today()
    logFileName = "./Logs/{today}.log".format(today=date)

    # Setup logging
    logging.basicConfig(filename=logFileName, filemode='a', level=logging.DEBUG,
                        format='%(asctime)s - %(levelname)s:%(message)s',
                        ↪  datefmt='%m/%d/%Y %I:%M:%S %p')

    nextStart = time.monotonic()
```

```
19      delta = 7.5  # 7.5 seconds between runs (time to perform all data collection
        ↪  and processing)

20

21      for i in range(8):
22          nextStart = nextStart + delta

23

24          # Classify code here
25          classData = classify.main()

26

27          # Allocation code here
28          allocate.main(classData)

29

30          # Wait until next classify time
31          if i != 7:
32              while nextStart > time.monotonic():
33                  time.sleep(0.1)
```

B.2   Classification

```
1   #!/usr/local/bin/python3.8
2   import datetime
3   import logging
4   import re
5   import subprocess

6

7

8   # Get bandwidth data from iftop and parse
9   def flow():
10      # Run iftop
11      # Arguments: -t, text mode (remove ncurses)
12      #            -c <file>, configuration input file
13      #            -s #, measure for # seconds
14      #            -i <network interface>, interface to listen on
```

```python
15      #               -L #, number of lines to display
16      #
17      # Redirect stderr to /dev/null
18      # Take stdout output and split each line into list
19      iftop = "iftop -t -c .iftoprc -s 3 -L 35 -i ens18"  # (FIXME: Change network
        ↪  interface to correct value)
20      proc_out = subprocess.run(args=iftop, shell=True, universal_newlines=True,
21                                 stdout=subprocess.PIPE, stderr=subprocess.DEVNULL)
22      top_list = proc_out.stdout.split("\n")
23
24      # Trim list to only contain relevant lines of data (data with per IP
        ↪  bandwidth)
25      data_list = []
26
27      for i in range(len(top_list)):
28          if re.search('^ {1,3}[0-9]', top_list[i]):
29              data_list.append(top_list[i])
30              data_list.append(top_list[i + 1])
31              i + 1
32
33      # Count = 2 * number of hosts
34      # Two lines per host (one upload, one download)
35      count = len(data_list)
36
37      # If list is empty, no data to work on
38      if count < 2:
39          return -1
40
41      # Dictionary to hold host information, in-coming, and out-going traffic
42      global host_dict
43      global host_list
44
45      host_dict = {}
```

```python
46         host_list = []
47
48         #
49         # For each host upload/download pair, extract information into format below
50         #
51         #     Host     |  Up Rate | Down Rate
52         #   <ip_addr>  |   Mbps   |   Mbps
53         #   <ip_addr>  |   Mbps   |   Mbps
54         #      ""      |    ""    |    ""
55         for i in range(int(count / 2)):
56             down_list = data_list[i * 2].split(" ")
57             up_list = data_list[(i * 2) + 1].split(" ")
58
59             while '' in up_list:
60                 up_list.remove('')
61
62             while '' in down_list:
63                 down_list.remove('')
64
65             host_ip = up_list[0]
66             up_rate = up_list[2]
67             down_rate = down_list[3]
68
69             # Standardize units
70             up_rate = unit(up_rate)
71             down_rate = unit(down_rate)
72
73             # Store data
74             host_data = [up_rate, down_rate]
75             host_dict[host_ip] = host_data
76             host_list.append(host_ip)
77
78
```

```python
79   # Classify each host's priority
80   def priority():
81
82       global prio_dict
83       prio_dict = {}
84
85       prio = 5
86
87       for ip in host_list:
88           bandwidth = host_dict[ip]
89
90           if bandwidth[0] < 200.0 and bandwidth[1] < 5000.0:
91               prio = 0
92           elif bandwidth[0] > 200.0 and bandwidth[1] < 5000.0:
93               prio = 1
94           elif bandwidth[0] < 200.0 and bandwidth[1] > 5000.0:
95               prio = 2
96           elif bandwidth[0] > 200.0 and bandwidth[1] > 5000.0:
97               prio = 3
98
99           prio_dict[ip] = prio
100
101          logging.info(" CLASSIFY -- Host: %s; Upload: %.2fKbps, Download: %.2fKbps,
     ↪  Priority: %d", ip, bandwidth[0], bandwidth[1], prio)
102
103
104  # Strip unit, standardize to Kbps
105  def unit(measure):
106      if "Mb" in measure:
107          ret = float(measure.strip("Mb")) * 1024
108          return ret
109      elif "Kb" in measure:
110          ret = float(measure.strip("Kb"))
```

```python
111        return ret
112    elif "b" in measure:
113        ret = float(measure.strip("b"))
114        return ret
115
116
117 def main():
118
119    outputDict = {}
120
121    if flow() != -1:
122        priority()
123        return host_dict
124
125    else:
126        logging.info(" CLASSIFY -- No data found")
127
128
129 if __name__ == '__main__':
130
131    date = datetime.date.today()
132    logFileName = "./Logs/{today}.log".format(today=date)
133
134    # Setup logging
135    logging.basicConfig(filename=logFileName, filemode='a', level=logging.DEBUG,
136                        format='%(asctime)s - %(levelname)s:%(message)s',
                     ↪  datefmt='%m/%d/%Y %I:%M:%S %p')
137
138    main()
```

B.3 Allocation

```python
#!/usr/local/bin/python3.8
import csv
import datetime
import logging
import os
import xml.etree.ElementTree as ET


# Standard amount by which bandwidth should be raised or lowered (Kbps)
incrementAmount = 2500
decrementAmount = 1000

# Individual Host max BW and min BW (Kbps)
minHost = 7500
maxHost = 100000

# Max bandwidth (Kbps)
maxNetworkBandwidth = 25000 * 30

configFile = 'config.xml'       # pfSense XML setting file (FIXME: Restore this
   ↪  path to '/conf/config.xml')
classifyFile = 'classData.csv'  # classify script output file


def readClassifyData():

    # Read data from classification script
    # Dictionary {Key: Value} where Key is <ip address> and Value is (Upload,
       ↪  Download)
    classifyData = {}

    with open(classifyFile) as csvFile:
```

```python
31            csvReader = csv.reader(csvFile)
32            for row in csvReader:
33                classifyData[row[0]] = (row[1], row[2])
34
35        return classifyData
36
37
38    def readXML():
39
40        # Get current download bandwidth allocations from limiters
41
42        tree = ET.parse(configFile)
43        root = tree.getroot()
44
45        currBW = {}
46
47        for queue in root.findall('./dnshaper/queue'):
48            ip_end = queue[0].text
49            if len(ip_end) > 5:  # Catch limiters not named _# (where # is last octet
                 ↪  of IP address)
50                continue
51            ip = "192.168.50.%s" % ip_end[1:]  # (FIXME: Restore start of IP address
                 ↪  to appropriate value)
52            bw = queue[5][0][0].text
53
54            currBW[ip] = bw
55
56        return currBW
57
58
59    def genBWList(currentAllocation, classifyData):
60
61        # Generate dictionary of IP : New Bandwidth Amount (In Kbps)
```

```python
62        newBWList = {}

63

64        for key in currentAllocation:

65

66            if classifyData is None or key not in classifyData:
67                # Assume no reading means no usage
68                if int(currentAllocation[key]) > minHost:
69                    newBWList[key] = str(int(currentAllocation[key]) -
                     ↪   decrementAmount)
70                    logging.info(" ALLOCATE -- Host: %s; Decrease cap, New Cap: %s",
                     ↪   key, newBWList[key])
71                continue

72

73            downloadVal = classifyData[key][1]

74

75            if downloadVal > (int(currentAllocation[key]) * 0.95) and
            ↪   int(currentAllocation[key]) < maxHost:
76                newBWList[key] = str(int(currentAllocation[key]) + incrementAmount)
77                logging.info(" ALLOCATE -- Host: %s; Increase cap, New Cap: %s", key,
                 ↪   newBWList[key])
78            elif downloadVal < (int(currentAllocation[key]) * 0.50) and
            ↪   int(currentAllocation[key]) > minHost:
79                newBWList[key] = str(int(currentAllocation[key]) - decrementAmount)
80                logging.info(" ALLOCATE -- Host: %s; Decrease cap, New Cap: %s", key,
                 ↪   newBWList[key])
81            else:
82                newBWList[key] = currentAllocation[key]
83                logging.info(" ALLOCATE -- Host: %s; No change, New Cap: %s", key,
                 ↪   newBWList[key])

84

85        return newBWList

86

87
```

```python
88   def writeXML(newBW):
89
90       # 1. Get new bandwidth amounts as input arg (dictionary)
91       # 2. Parse /conf/config.xml and find each queue's IP
92       # 3. Use IP as key in dictionary to get new bandwidth value
93       # 4. Write out new XML file
94
95       tree = ET.parse(configFile)
96       root = tree.getroot()
97
98       for queue in root.findall('./dnshaper/queue'):
99           ip_end = queue[0].text
100          if len(ip_end) > 5:  # Catch limiters not named _# (where # is last octet
              ↪ of IP address)
101              continue
102          ip = "192.168.50.%s" % ip_end[1:]  # (FIXME: Restore start of IP address
              ↪ to appropriate value)
103          try:
104              queue[5][0][0].text = newBW[ip]
105          except KeyError:  # Skip queue if corresponding IP is not in dictionary
106              continue
107
108      tree.write(configFile)
109
110      return 0
111
112
113  def reloadFirewall():
114
115      # 1. Remove /tmp/config.cache (Reloads config file)
116      # 2. Run /etc/rc.filter_configure (Reloads pfsense firewall)
117
118      os.system('rm /tmp/config.cache')
```

```python
119        os.system('/etc/rc.filter_configure')

120

121        return 0

122

123

124   def main(classData):

125

126        # 1. Get and store current bandwidth allocations

127        # 3. Generate list of new bandwidth allocations

128        # 4. Write new allocation parameters out to XML File

129        # 5. Reload firewall

130

131        currAllots = readXML()

132        newBW = genBWList(currAllots, classData)

133        writeXML(newBW)

134        reloadFirewall()

135

136        return 0

137

138

139   if __name__ == '__main__':

140

141        date = datetime.date.today()

142        logFileName = "./Logs/{today}.log".format(today=date)

143

144        # Setup logging

145        logging.basicConfig(filename=logFileName, filemode='a', level=logging.DEBUG,

146                        format='%(asctime)s - %(levelname)s:%(message)s',

                 ↪    datefmt='%m/%d/%Y %I:%M:%S %p')

147

148        main({})
```

Appendix C

ARGoS-EMANE Interface Source Code

C.1   Main Code

```python
#!/usr/bin/env python3

import os
import signal
import struct
import sys
import time
from multiprocessing import shared_memory

from emane.events import EventService, LocationEvent

from libs.structs import *
from libs.drone import EMANEDrone

META_FORM = 'HHdIIddd'
META_SIZE = struct.calcsize(META_FORM)
POSE_FORM = 'Hddd'
```

```
18  POSE_SIZE = struct.calcsize(POSE_FORM)

19  COMM_FORM = 'HHdd'

20  COMM_SIZE = struct.calcsize(COMM_FORM)

21

22  EMANE_PID = os.getpid()

23  GW_ID = 0 # TODO: Select a constant ID for the gateway (Initialize a translation
    ↪   table between EMANE IDs and ARGoS IDs?)

24

25  def handler_sigcont(sig, frame):

26      return

27

28  def handler_sigterm(sig, frame):

29      shm_meta.close()

30      shm_pose.close()

31      shm_comm.close()

32      sys.exit(0)

33

34

35  def wait_for_argos():

36      time.sleep(0.1)

37      os.kill(sys_meta.argos_pid, signal.SIGCONT)

38      print("Sending SIGCONT to " + str(sys_meta.argos_pid))

39      signal.raise_signal(signal.SIGSTOP)

40

41

42  def translateID(ARGoS_ID):

43      pass

44

45  def init():

46      signal.signal(signal.SIGTERM, handler_sigterm)

47      signal.signal(signal.SIGCONT, handler_sigcont)

48

49      global shm_meta
```

```python
50        global shm_pose
51        global shm_comm
52
53        shm_meta_exists = False
54        while(not shm_meta_exists):
55            try:
56                shm_meta = shared_memory.SharedMemory(name="argos_emane_meta",
                  ↪  create=False, size=META_SIZE)
57            except FileNotFoundError:
58                time.sleep(5)
59                continue
60            shm_meta_exists = True
61
62        global sys_meta
63        sys_meta = RobotMeta()
64        sys_meta.unpack(shm_meta)
65        sys_meta.emane_pid = EMANE_PID
66        sys_meta.pack(shm_meta)
67
68        print("ARGoS found continuing setup")
69
70        shm_pose = shared_memory.SharedMemory(name="argos_emane_pose", create=False,
          ↪  size=POSE_SIZE*sys_meta.num_drone)
71        shm_comm = shared_memory.SharedMemory(name="argos_emane_comms", create=False,
          ↪  size=COMM_SIZE*sys_meta.num_comms)
72
73        global drone_nodes
74        drone_nodes = [EMANEDrone() for i in range(sys_meta.num_drone)]
75
76        global robotpose
77        global robotcomm
78        robotpose = [RobotPose() for i in range(sys_meta.num_drone)]
79        robotcomm = [RobotComms() for i in range(sys_meta.num_comms)]
```

```
80
81     print("Setting up gateway node")
82     gateway_service = EventService(('224.1.2.8', 45703, 'control0')) # These
   ↪    values come from EMANE config files
83     gateway_loc = LocationEvent()
84     gateway_loc.append(GW_ID, latitude=sys_meta.gw_lat, longitude=sys_meta.gw_lon,
   ↪    altitude=sys_meta.gw_alt, yaw=0, pitch=0, roll=0)
85     gateway_service.publish(0, gateway_loc)
86
87
88 def update_drone():
89     global shm_meta
90     global shm_pose
91     global robotpose
92
93     prev_drone = sys_meta.num_drone
94     sys_meta.unpack(shm_meta)
95
96     # NOTE: This should theoretically never fire (without EMANE being configured
   ↪    to start with more nodes than ARGoS)
97     #        EMANE can not create new nodes during runtime
98     if(prev_drone < sys_meta.num_drone):
99         shm_pose.close()
100        shm_pose = shared_memory.SharedMemory(name="argos_emane_pose",
   ↪        create=False)
101        robotpose = [RobotPose() for i in range(sys_meta.num_drone)]
102
103    for i in range(sys_meta.num_drone):
104        buf = shm_pose.buf[i*POSE_SIZE:(i+1)*POSE_SIZE]
105        robotpose[i].unpack(buf)
106        drone_nodes[i].id = robotpose[i].id
107        drone_nodes[i].lat = robotpose[i].lat
108        drone_nodes[i].lon = robotpose[i].lon
```

```python
109             drone_nodes[i].alt = robotpose[i].alt
110
111         print("Sending location events to EMANE")
112         service = EventService(('224.1.2.8', 45703, 'control0')) # These values come
        ↪   from EMANE config files
113         for drone in drone_nodes:
114             drone.location_event(service)
115
116
117     def communicate():
118         global shm_meta
119         global shm_comm
120         global robotcomm
121
122         prev_comms = sys_meta.num_comms
123         sys_meta.unpack(shm_meta)
124
125         if(prev_comms < sys_meta.num_comms):
126             shm_comm.close()
127             shm_comm = shared_memory.SharedMemory(name="argos_emane_comms",
            ↪   create=False)
128             robotcomm = [RobotComms() for i in range(sys_meta.num_comms)]
129
130         print(sys_meta)
131         # BUG: Correlate list index with drone id (breaks if drone poses or drone IDs
        ↪   are not in sequential order)
132         try:
133             for i in range(sys_meta.num_comms):
134                 buf = shm_comm.buf[i*COMM_SIZE:(i+1)*COMM_SIZE]
135                 robotcomm[i].unpack(buf)
136                 # drone_nodes[0].inc_buffer(robotcomm[i].buff_size)
137                 drone_nodes[i].buff = robotcomm[i].buff_size # NOTE:
138                 robotcomm[i].sent = drone_nodes[i].do_broadcast()
```

```
139                buf = robotcomm[i].pack()
140                shm_comm.buf[i*COMM_SIZE:(i+1)*COMM_SIZE] = buf
141
142                # TODO: Check latency between all pairs and store to report to SySML
143        except:
144            pass
145
146  if __name__=='__main__':
147      iternum = 0
148
149      init()
150      print(sys_meta)
151      while True:
152          wait_for_argos()
153          time.sleep(1)
154          update_drone()
155          communicate()
156
157          print(iternum)
158          iternum+=1
```

C.2 Shared Memory Data Structures

```
1   import struct
2   from ctypes import *
3   from dataclasses import dataclass
4
5   @dataclass
6   class RobotMeta():
7       """
8       Struct containing metadata shared between ARGoS and EMANE
9
10      Attributes
```

```
11          ----------
12          num_drone - Number of active robots
13          num_comms - Number of robots attmempting to broadcast
14          deltaT    - Time per step
15          argos_pid - ARGoS Process ID
16          emane_pid - EMANE Process ID (PID of Interface Script, not EMANE itself)
17          gw_lat    - Latitude coordinate of gateway node
18          gw_lon    - Longitude coordinate of gateway node
19          gw_alt    - Altitude (meters) of gateway node
20          struct_format - Metadata used to unpack data from shared memory
21          """
22          num_drone: c_ushort = None
23          num_comms: c_ushort = None
24          deltaT:    c_double = None
25          argos_pid: c_uint = None
26          emane_pid: c_uint = None
27          gw_lat:    c_double = None
28          gw_lon:    c_double = None
29          gw_alt:    c_double = None
30
31          __struct_format: str = 'HHdIIddd'
32
33          def unpack(self, shm):
34              self.num_drone, self.num_comms, self.deltaT, self.argos_pid,
            ↪   self.emane_pid, self.gw_lat, self.gw_lon, self.gw_alt \
35                  = struct.unpack(self.__struct_format, shm.buf.tobytes())
36
37          def pack(self, shm):
38              buf = struct.pack(self.__struct_format, self.num_drone, self.num_comms,
            ↪   self.deltaT, self.argos_pid, self.emane_pid, self.gw_lat, self.gw_lon,
            ↪   self.gw_alt)
39              shm.buf[:struct.calcsize(self.__struct_format)] = buf
40
```

```python
41
42  @dataclass
43  class RobotPose():
44      """
45      Struct containing location of each robot
46
47      Attributes
48      ----------
49      id - Robot ID
50      lat - Latitude of robot
51      lon - Longitude of robot
52      alt - Altitude of robot
53      struct_format - Metadata used to unpack data from shared memory
54      """
55      id: c_ushort = None
56      lat: c_double = None
57      lon: c_double = None
58      alt: c_double = None
59      __struct_format: str = 'Hddd'
60
61      def unpack(self, buf):
62          self.id, self.lat, self.lon, self.alt =
            ↪   struct.unpack(self.__struct_format, buf)
63
64      def pack(self):
65          buf = struct.pack(self.__struct_format, self.id, self.lat, self.lon,
            ↪   self.alt)
66          return buf
67
68
69  @dataclass
70  class RobotComms():
71      """
```

```
72        Struct containing the communication data for each robot
73
74        Attributes
75        ----------
76        id_from - Robot ID Transmitting
77        id_to   - Robot ID Receiving
78        buff_size - Size of data robot wants to transmit (bytes)
79        sent - Size of data actually transmitted (bytes)
80        struct_format - Metadata used to unpack data from shared memory
81        """
82        id_from: c_ushort = None
83        id_to: c_ushort = None
84        buff_size: c_double = None
85        sent: c_double = None
86        __struct_format: str = 'HHdd'
87
88        def unpack(self, buf):
89            self.id_from, self.id_to, self.buff_size, self.sent =
            ↪   struct.unpack(self.__struct_format, buf)
90
91        def pack(self):
92            buf = struct.pack(self.__struct_format, self.id_from, self.id_to,
            ↪   self.buff_size, self.sent)
93            return buf
```

C.3   Drone Object

```
1  from dataclasses import dataclass
2  from emane.events import *
3  from ctypes import *
4
5  @dataclass
6  class EMANEDrone():
```

```python
      '''
      Corresponds to a single drone node
      Holds its own ID, Location, and TxBufferSize

      Attributes
      ----------
      id - Drone node ID
      lat - Node's latitude
      lon - Node's longitude
      alt - Node's altitude
      buff - Number of bytes this node needs to still transmit
      sent - Number of bytes this node sent this iteration
      '''
      id:   c_ushort = None
      lat:  c_double = None
      lon:  c_double = None
      alt:  c_double = None
      buff: c_double = 0.0

      def inc_buffer(self, size):
          self.buff += size

      def dec_buffer(self, size):
          self.buff -= size


      def location_event(self, service):
          loc_event = LocationEvent()
          loc_event.append(self.id, latitude=self.lat, longitude=self.lon,
          ↪   altitude=self.alt, yaw=0, pitch=0, roll=0)
          service.publish(0, loc_event)
          return 0
```