

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	vi
List of Tables	viii
1 Introduction	1
2 Overview on Network Emulation	3
2.1 Testing Communication Networks	3
2.2 Evaluation of Network Testing Tools	4
2.3 Using EMANE	9
2.3.1 Emulation Model Processing	12
2.3.2 Transport Boundary Processing	16
2.3.3 Event Processing	17
2.4 Routing in Mobile Mesh Networks	19
2.4.1 Proactive Mesh Routing	20
2.4.2 Reactive Routing	21
2.5 Chapter Summary	22
3 Hybrid Wireless Rural Broadband Networks	23
3.1 Hardware Testbed Network Topologies	23
3.1.1 OVERCOME Testbed	24
3.1.2 ZoomTEL Testbed	25
3.2 Creating the Networks in EMANE	26
3.3 Hardware Results versus Emulation Results	28
3.4 Chapter Summary	31
4 Networking Software Development Environment	32
4.1 Intelligent Method of Bandwidth Distribution	32
4.2 Implementing the Software	34
4.2.1 Stage 1: Classification	34
4.2.2 Stage 2: Allocation	37
4.3 Effectiveness of the Program	38
4.4 Chapter Summary	40

5	Dynamic Robot Swarm Networks	42
5.1	Extending Existing Software	42
5.2	Integrating the Software	43
5.3	Integration Design Decisions	46
5.4	Integration Results	47
5.5	Chapter Summary	48
6	Conclusion	49
6.1	Evaluating Network Emulation	49
6.2	Research Outcomes	50
6.3	Future Work	51
	Bibliography	52
A	Installation of EMANE	56
B	Intelligent Router Source Code	58
B.1	Control Script	58
B.2	Classification	59
B.3	Allocation	63
C	ARGoS-EMANE Interface Source Code	68
C.1	Main Code	68
C.2	Shared Memory Data Structures	73
C.3	Drone Object	75

List of Figures

2.1	An example of the OMNeT++ Simulator GUI.	6
2.2	An example of the CORE Emulator GUI.	7
2.3	The configuration menu for EMANE, within CORE. From [1].	8
2.4	Downloading the precompiled EMANE binaries using the <i>wget</i> command.	10
2.5	Installing the EMANE program <i>dpkg</i> command.	10
2.6	Verifying EMANE installed correctly by displaying the version number.	10
2.7	An overview of an individual EMANE emulated network node.	12
2.8	A generic configuration file for an EMANE PHY Plugin	15
2.9	The packet complete rate (PCR) table and corresponding curve.	16
2.10	An example EMANE topology using the raw transport plugin.	17
2.11	An example EMANE topology using the virtual transport plugin.	18
2.12	An example of an EEL file provided to the EMANE event service.	19
2.13	An overview of an individual EMANE emulated network node.	21
3.1	The network topology of the testbed built as part of Project OVERCOME in Turney, MO.	24
3.2	The network topology of the experimental testbed built as part of the ZoomTEL project.	25
3.3	The emulation testbed topology corresponding to the OVERCOME project.	27
3.4	The emulation testbed topology corresponding to the ZoomTEL project.	27
3.5	The console output of an <i>iperf3</i> test used to find the throughput for part of the ZoomTEL EMANE testbed.	29
4.1	The modified OVERCOME EMANE testbed used for development of the intelligent router program. The primary modification is the removal of the mmWave environment.	33
4.2	An example of the output of the <i>iftop</i> utility used to measure bandwidth on the OVERCOME network.	35
4.3	An example of the data output but the classification stage of the intelligent router program. IP addresses have been censored for privacy.	35
4.4	An overview of the algorithm that operates to classify each host on the network as a part of the intelligent router program.	36

4.5	An overview of the algorithm that operates to allocate bandwidth to each host on the network as a part of the intelligent router program.	38
4.6	The total network usage for thirty houses in the Project OVERCOME testbed during weeks 2 and 3 of the intelligent router test.	40
4.7	The total network usage for thirty houses in the Project OVERCOME testbed during weeks 2 and 3 of the intelligent router test.	41
5.1	The topology of the ARGoS-EMANE integration system. All the major systems as well as the interconnections between them are displayed.	45
5.2	A dump of several events that were sent over the EMANE event channel. These location events show that ARGoS is delivering location data to the interface script, which is subsequently delivering location events to EMANE.	48

List of Tables

2.1	Pros and Cons of Different Types of Network Testing	4
2.2	Overview of Advantages and Disadvantages of Different Networking Testing Tools	8
3.1	mmWave Model EMANE Parameters	26
3.2	Ubiquiti LTU Model EMANE Parameters	26
3.3	Latency and throughput measurements from the Projects OVERCOME, ZoomTEL, and EMANE testbeds.	30
4.1	Priority groups for the intelligent router, based on user bandwidth behaviors	36
5.1	Contents of the shared memory metadata file. Includes which processes are responsible for what data	44
5.2	Contents of the shared memory robot pose file. Includes which processes are responsible for what data	44
5.3	Contents of the shared memory robot communications file. Includes which processes are responsible for what data	44

Chapter 1

Introduction

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 becomes apparent. Despite this need for interconnectivity, there is still a large divide in the amount of people with access to broadband services. A study conducted in early 2021 found that rural broadband deployments are increasing, but rural communities still lag behind suburban and urban communities in terms of connectivity [2]. Additionally over the last 10 years, the number of U.S. adults that rely on the Internet has grown by 10% [3]. This heavy reliance on networks and the Internet is expected to continue, with experts estimating in the next 10 years, the number of IoT devices alone will triple from ten billion to thirty billion [4]. The demand for new wireless technologies and systems creates issues with testing and deployment. Part of the reason rural communities lag behind is due to the difficulty in rapidly developing, deploying, and testing communication technology. To attempt to minimize this issue several tools have been created that allow for testing networks, with the goal of lowering the difficulty of testing.

Many software and combination software-hardware platforms exist for testing networks. Tools like ns-3, MATLAB, and GNURadio all provide different platforms for testing with different areas of focus. MATLAB is a good tool for doing signal processing and analysis, but has little support for real-time networking. ns-3 provides great support for network protocols, but abstractions are made at the physical layer that may call into question the results in many use cases.

Despite several tools existing to test networks, the issue of cost, accessibility, and accuracy are still rampant. Many of the tools that have highly accurate models of the entire

OSI model, from the physical layer through the transport layer, are very expensive and not attainable. NetSim and OPNET (now part of the Riverbed platform), provide enterprise-grade modeling, but requires licensing to use. GNURadio allows for interactions with wireless communication hardware and software-defined radios, but the hardware component is still an expensive cost that needs to be avoided. Another issue with several of the existing tools is the complexity to set up. Many tools require extensive programming knowledge to be able to achieve the desired result, or do not provide an easy mechanism with which to analyze results.

This thesis proposes several solutions to contribute to the field of network and communications testing:

- 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. This is used as an example of how EMANE can be used as a network software development environment.
- Basic integration between EMANE and the robot swarm simulator ARGoS is created. EMANE is shown to be capable of extending and enhancing other tools to provide accurate communication emulation when required.

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 in which we can use the tool, we must first understand 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 tool's installation, configuration, and operation is presented. Finally, a brief overview of mobile ad-hoc network (MANET) routing 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 are done. Hardware, however, is expensive to purchase, time-consuming to deploy, and often difficult to troubleshoot if errors do not consistently appear [6]. These factors make hardware testing not accessible to users that have a low budget.

Network simulation is one solution to testing that solves many issues with testing on hardware. Several free and open-source network simulators like ns-3 [7] or OMNeT++ [8] are commonly used and provide a solution to the high hardware costs. 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 events to happen. However, the caveat is that simulation models must be highly accurate when developed, or 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 confirm the accuracy of the models before they 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. However, emulators operate on actual network data instead of modeling the behavior of a network. Because emulation testbeds operate on actual network traffic, they also can interface with hardware. This hardware-in-the-loop functionality allows hardware testing without the need to build an entire hardware network. This characteristic of operating on real network traffic also has the downside of introducing more computational overhead. The system emulating the network needs to manage all the test traffic as well as the effects and permutations that are imparted on the traffic.

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

Testbed Type	Pros	Cons
Hardware	<ul style="list-style-type: none"> • Highly accurate • Does not require modification of networking software 	<ul style="list-style-type: none"> • Expensive to build • Time consuming to deploy and configure • Errors can be sporadic
Simulation	<ul style="list-style-type: none"> • Free tools available • Can run faster than real-time • Easy to reconfigure and modify 	<ul style="list-style-type: none"> • Models must be designed to be highly accurate • Software must be translated to a simulation model
Emulation	<ul style="list-style-type: none"> • Free tools available • Can run native implementations of network software • Can interface with hardware 	<ul style="list-style-type: none"> • Must run in real-time • Requires higher computational overhead

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

2.2 Evaluation of Network Testing Tools

During the initial research for this thesis, several network simulation and emulation tools were considered for use in this thesis. 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 by introducing EMANE and explaining the driving reasons for its selection.

One of the most common tools, ns-3, is a discrete, event-based network simulation tool commonly used to simulate TCP/IP networks [7]. It is the successor to the ns-2 tool [10], with the two significant differences being how simulation scripts are written and executed in the simulator. In ns-3, the core simulator and experiment scripts are written in C++. ns-3 also provides Python bindings so smaller scripts can be written using Python instead. This makes ns-3 much simpler than ns-2, which required the use of Tcl scripts to create experiments [11]. The migration from ns-2's programming method to ns-3's makes the tool easier to use. However, it still requires a lot of programming knowledge to understand how to utilize the library of models provided. Being one of the most commonly used tools, ns-3's library of simulation modules has been extensively validated, but these modules typically focus on the network layer and above. ns-3 can be used as an emulator; however, the motivation for the emulation mode in ns-3 was primarily driven by the ability to use ns-3 with hardware [12]. Since we would like to avoid wireless hardware 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 explicitly 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. 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 modules were developed under [13]. One of the advantages of OMNeT++ is that it supports a graphical user interface that allows for more accessible building of and interaction with network testbeds. Figure 2.1 shows an example of this GUI, which is based on the Eclipse IDE.

Since OMNeT++ is not designed for communications networks specifically and many plugins are incompatible, it was decided not to be used.

The Common Open Research Emulator (CORE) is an emulation tool focused on the emulation of layers three (network layer) and above in the OSI stack [14]. Of all the

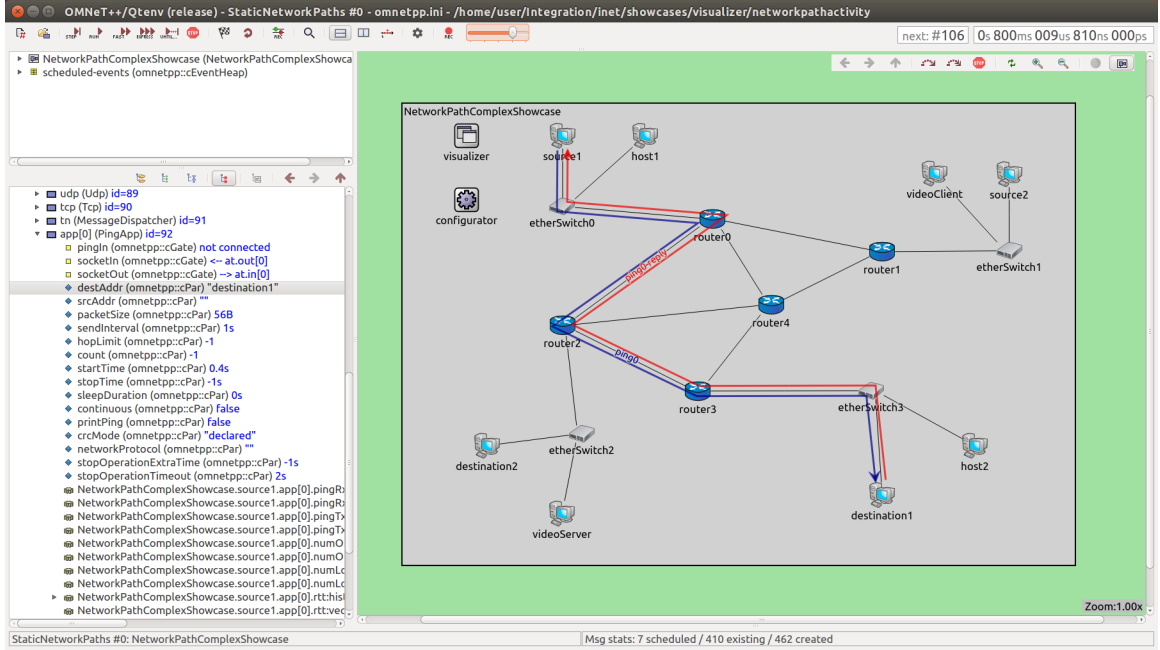


Figure 2.1: An example of the OMNeT++ Simulator GUI.

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, into a network implementation. Figure 2.2 is a basic example of how a typical CORE session appears. CORE also allows users to create virtual nodes via a Python framework, which is helpful 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 if nodes are close enough to communicate, they have a perfect connection, and if they are far enough apart by a certain distance, they can not communicate. This simplification is not acceptable for most testing requiring accurate wireless communication modeling. So to solve this issue, EMANE was integrated into CORE so that all physical and data link layer emulation happened through EMANE instead [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 is that most 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.

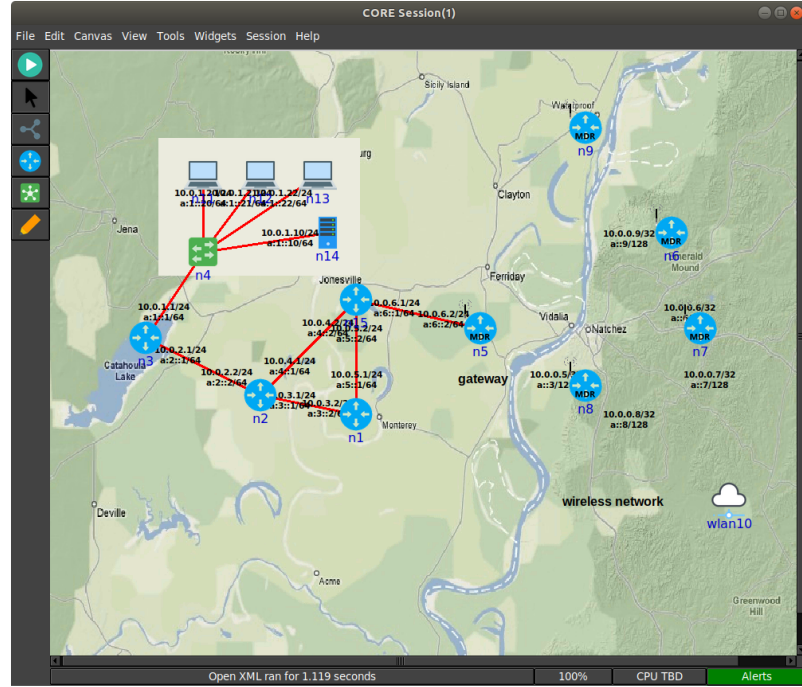


Figure 2.2: An example of the CORE Emulator GUI.

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, therefore 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 solve the initial complexity issue. Despite all this, once the user forms a solid understanding of the tools and systems within the software, it can be used to effectively and quickly create wireless networks. EMANE’s ability to be configured through XML files makes deployment of networks very rapid, and because the included models are

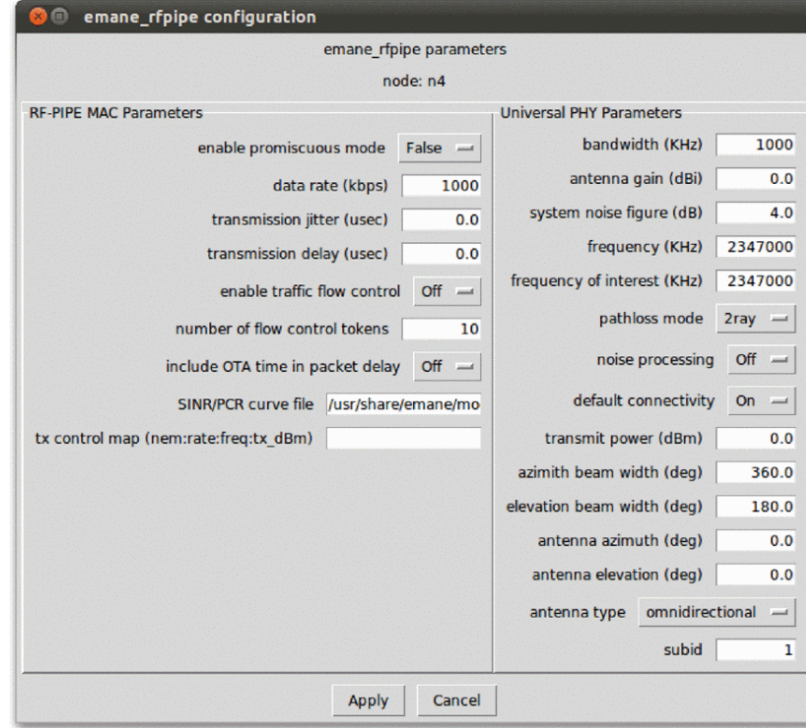


Figure 2.3: The configuration menu for EMANE, within CORE. From [1].

pluggable, these configuration files can be reused in other networks using the same wireless technology. For these reasons and the reasons highlighted previously regarding other testing tools, EMANE was selected as the tool of choice for this thesis. The next section is dedicated to understanding how to install the tool, and how all the pieces work together.

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

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

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. The primary two methods of installation are using the bundle of pre-built binaries provided by AdjacentLink 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 in this thesis, the precompiled bundle is sufficient for our purposes. 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 this version supports the Rocky 8, Fedora 37, and Ubuntu 20.04 Linux distributions.

These instructions assume a fresh installation of Ubuntu 20.04.4 is being used and that the 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 exact commands will differ. Refer to [16, 17] for further details. See Appendix A for a list of the specific commands used 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.



```

emane@EMANEDemo: ~$ wget https://adjacentlink.com/downloads/emane/emane-1.3.3-release-1.ubuntu-20_04.amd64.tar.gz
--2022-12-15 10:09:03-- https://adjacentlink.com/downloads/emane/emane-1.3.3-release-1.ubuntu-20_04.amd64.tar.gz
Resolving adjacentlink.com (adjacentlink.com)... 192.155.90.41
Connecting to adjacentlink.com (adjacentlink.com)|192.155.90.41|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 14483104 (14M) [application/x-gzip]
Saving to: 'emane-1.3.3-release-1.ubuntu-20_04.amd64.tar.gz'

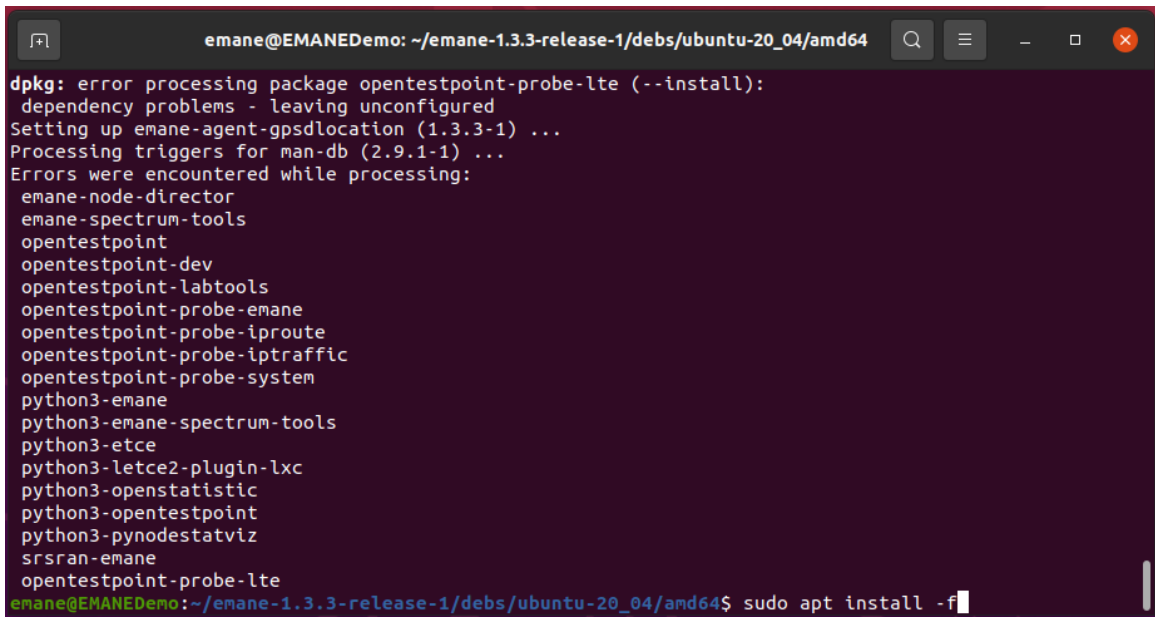
emane-1.3.3-release-1.ubuntu- 100%[=====] 13.81M 22.9MB/s in 0.6s

2022-12-15 10:09:04 (22.9 MB/s) - 'emane-1.3.3-release-1.ubuntu-20_04.amd64.tar.gz' saved [14483104/14483104]

emane@EMANEDemo: ~$

```

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

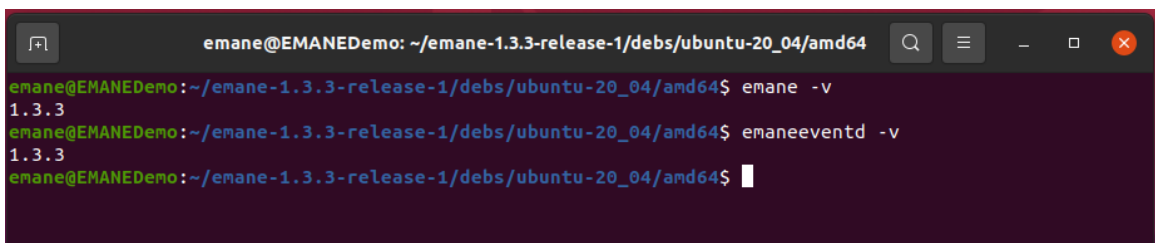


```

emane@EMANEDemo: ~/emane-1.3.3-release-1/debs/ubuntu-20_04/amd64
dpkg: error processing package opentestpoint-probe-lte (--install):
 dependency problems - leaving unconfigured
Setting up emane-agent-gpsdlocation (1.3.3-1) ...
Processing triggers for man-db (2.9.1-1) ...
Errors were encountered while processing:
 emane-node-director
 emane-spectrum-tools
 opentestpoint
 opentestpoint-dev
 opentestpoint-labtools
 opentestpoint-probe-emane
 opentestpoint-probe-iproute
 opentestpoint-probe-iptraffic
 opentestpoint-probe-system
 python3-emane
 python3-emane-spectrum-tools
 python3-etce
 python3-letce2-plugin-lxc
 python3-openstatistic
 python3-opentestpoint
 python3-pynodestatviz
 srsran-emane
 opentestpoint-probe-lte
emane@EMANEDemo: ~/emane-1.3.3-release-1/debs/ubuntu-20_04/amd64$ sudo apt install -f

```

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



```

emane@EMANEDemo: ~/emane-1.3.3-release-1/debs/ubuntu-20_04/amd64
emane@EMANEDemo: ~/emane-1.3.3-release-1/debs/ubuntu-20_04/amd64$ emane -v
1.3.3
emane@EMANEDemo: ~/emane-1.3.3-release-1/debs/ubuntu-20_04/amd64$ emaneeventd -v
1.3.3
emane@EMANEDemo: ~/emane-1.3.3-release-1/debs/ubuntu-20_04/amd64$

```

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

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 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 number 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 (LXC)s 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 situations and will be examined in depth in the next three subsections.

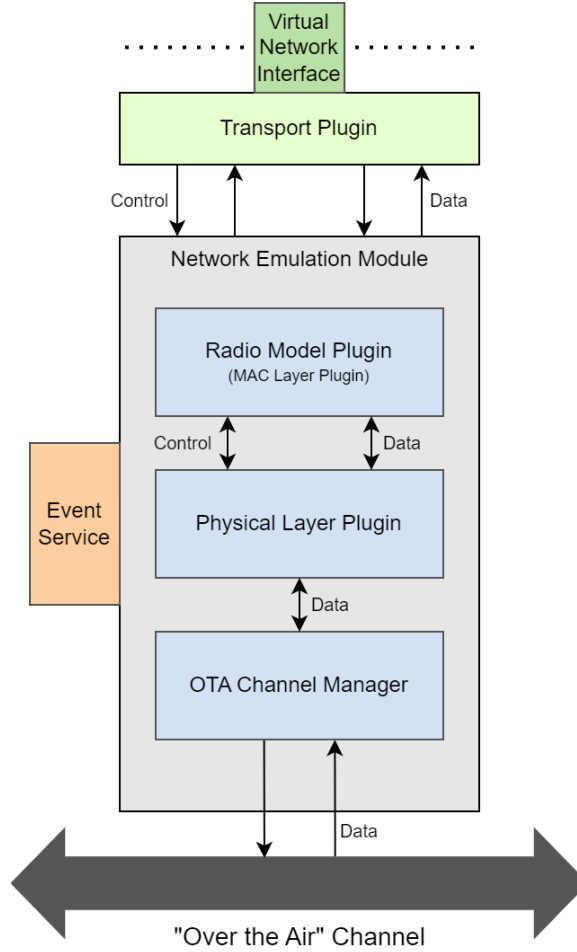


Figure 2.7: An overview of an individual EMANE emulated network node.

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 in 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 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 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 only one PHY model is included in EMANE by default. This model is responsible for attributes like pathloss and signal propagation, fading, noise modeling, and 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>
```

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 actively transmitting, and adding it to the noise floor of any receiving node if that receiver's set 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.

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

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

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 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 rfPipe 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

```

<?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

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 decide if the packet should be lost or otherwise corrupted due to noise.

One of the essential pieces of the emulation processing system is ensuring the models used are accurate to hardware, to ensure results 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

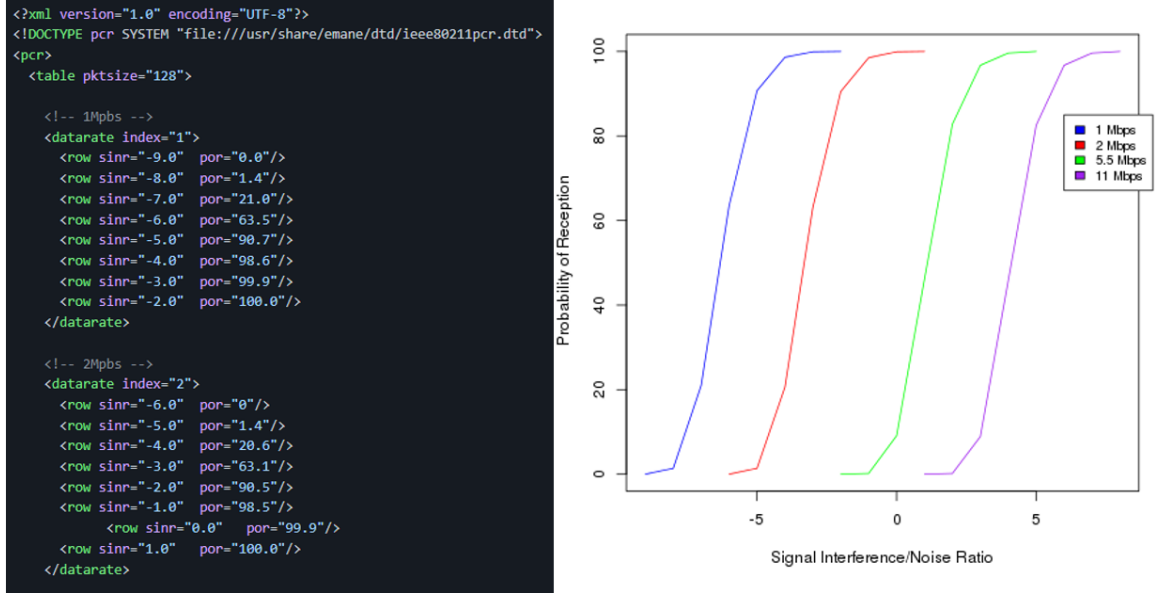


Figure 2.9: The packet complete rate (PCR) table and corresponding curve.

as for any custom models created. For the purposes of this thesis, EMANE is being used to test 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 handling 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 LXC's, 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.

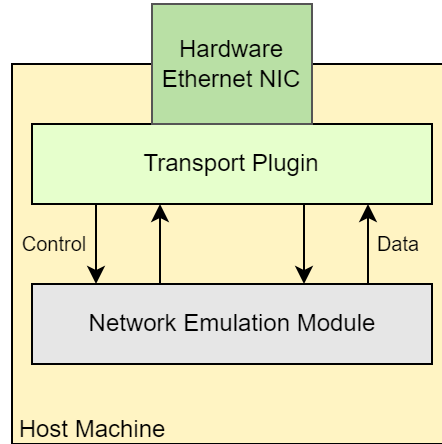


Figure 2.10: An example EMANE topology using the raw transport plugin.

The second type of transport are virtual transport plugins. These are more the commonly used plugins that carry data from the emulator instance to application space via a virtual network interface. When EMANE is started in 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 is bridged together with all the other nodes to allow traffic to pass. Figure 2.11 shows an example layout of the virtual network interfaces bridged together between EMANE instances, internal on the host machine.

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

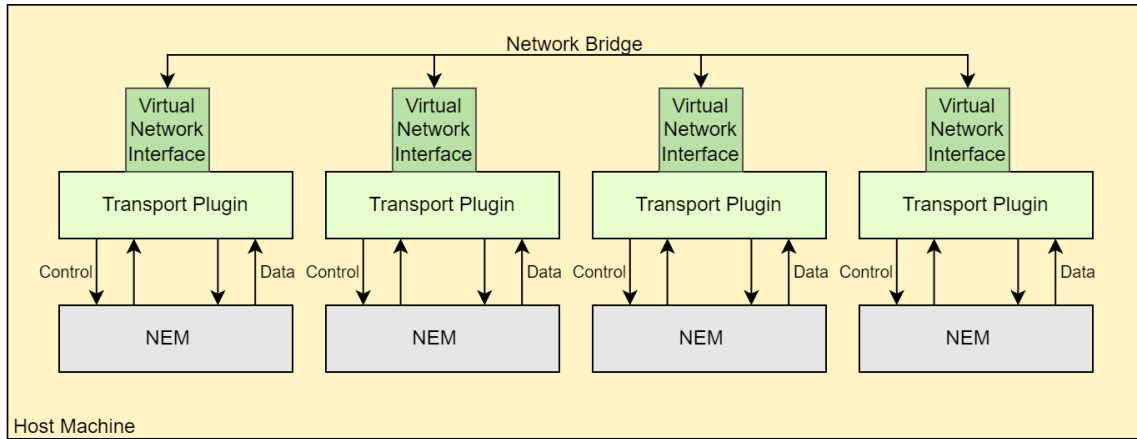


Figure 2.11: An example EMANE topology using the virtual transport plugin.

primarily influences pathloss values that are calculated internally to the tool when using *freespace* or *2ray* settings. The location event can take 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 by which 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 for that event type. 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 virtual 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 r
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 r
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:1,11.050000,22.000330,11.000000,13.000000,46,55 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 originally to work with mobile ad-hoc networks, also known as MANETs. While emulating other network types is possible, MANET routing protocols work well with EMANE's architecture and are often used. 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 intermittently 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 create instability in the network.

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 identifying the link that first delivers an OGM packet from a new node. That identified link is then flagged as the best way to send a packet to the node indicated in the new 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.

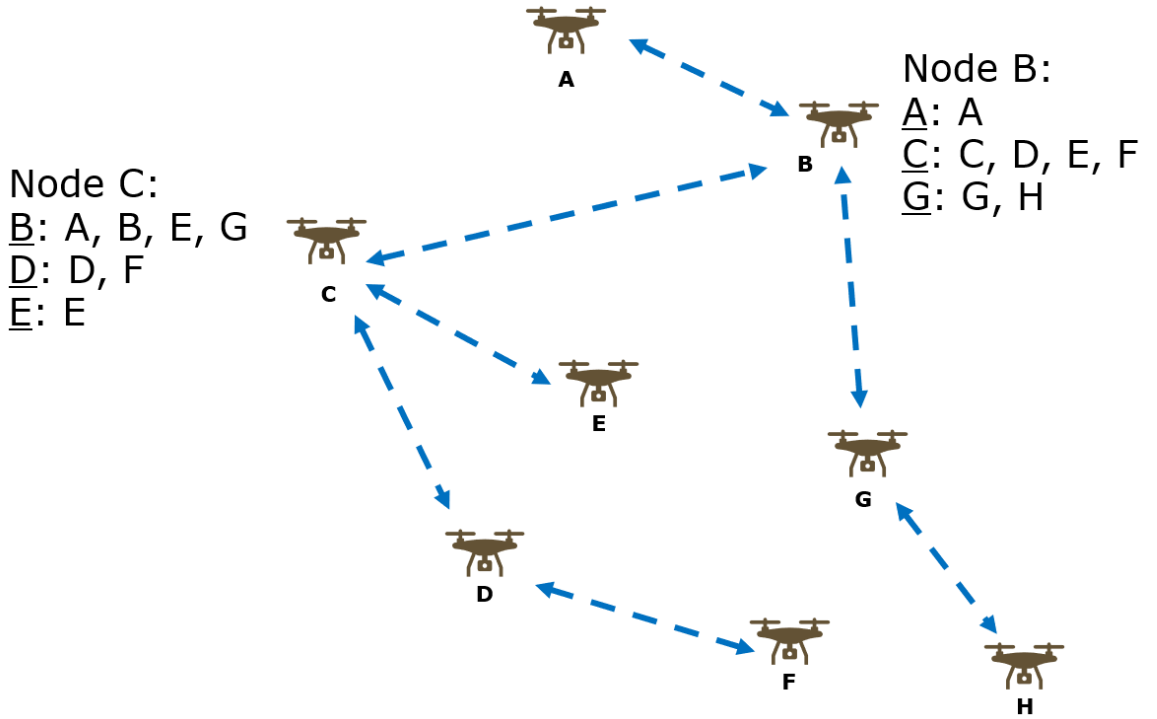


Figure 2.13: An overview of an individual EMANE emulated network node.

2.4.2 Reactive Routing

The other 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 respond, and once a full route is found the traffic is sent. The route is then stored and considered a good route for that destination until an attempt at using the route fails. At that point the discovery process is repeated to find 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 individually 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 created EMANE testbeds.

Chapter 3

Hybrid Wireless Rural Broadband Networks

The first testing scenario EMANE was placed under was emulating two wireless, hybrid network topologies that were designed to more effectively deliver broadband to small rural communities. Since deploying broadband to rural communities can be expensive through traditional methods, wireless distribution topologies have become a popular way of bringing the Internet to these communities. As was explained, testing wireless networks in hardware is rather expensive, and finding a location that meets the environmental factors of the target rural communities is challenging. To solve these issues, initial testing can be conducted in EMANE. This allows an understanding of the interactions between the technologies selected to be formed, and do basic validation that the proposed network can feasibly achieve its goal.

3.1 Hardware Testbed Network Topologies

Both of the network architectures that will be examined in this chapter were eventually set up in hardware. While these hardware testbeds were being built, emulated EMANE testbeds were also set up. Both testbeds can be classified by two distinct wireless technologies, the first stage is a mmWave backhaul. This wireless, high data rate backhaul was intended to replace the traditional wired backhaul, that makes reaching these rural communities so difficult. The other technology used was Ubiquiti's proprietary LTU protocol. The LTU leg of the network consisted of the distribution portion of the network and

was responsible for bridging the connected houses to the backhaul. By abstracting both of these technologies into EMANE, testing characteristic behaviors of the network with several hosts was possible. A pfSense Router was also used in both testbeds to handle all routing requirements. Thanks to EMANE’s hardware in the loop capabilities, the actual pfSense software was used in the experiment emulation testbed.

3.1.1 OVERCOME Testbed

The first of the two testbeds was located in Turney, Missouri, a small community outside of Kansas City that was not currently covered by any of the Internet Service Providers in the area. The topology of the network can be seen in Figure 3.1. In order to create this

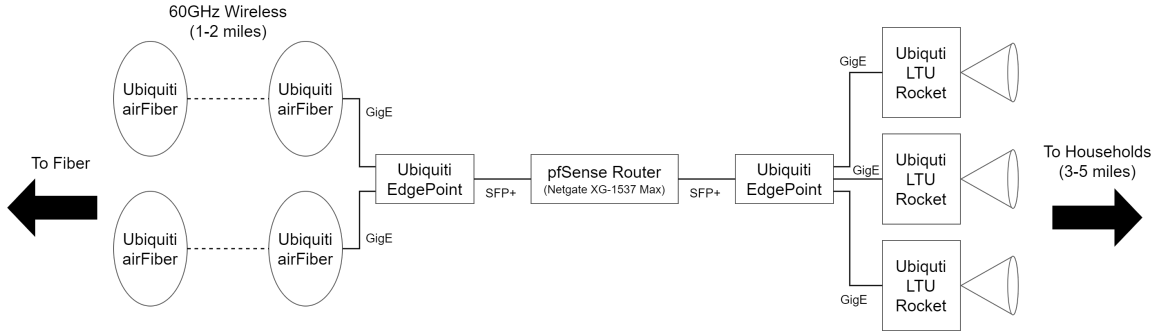


Figure 3.1: The network topology of the testbed built as part of Project OVERCOME in Turney, MO.

topology in EMANE, we first had to understand the major characteristics of the hardware being mimicked. The mmWave backhaul consisted of four Ubiquiti airFiber 60LR radios. These two pairs of point-to-point radios acted as the backhaul of the network and carried the traffic from the nearest location with fiber, to the center of the community. Once arriving at the center of Turney, the traffic was passed through a commercial Netgate XG-1537 Max router running pfSense. This router was primarily responsible for switching traffic off the ISP’s network and onto the last leg of the network to the homes. The final piece of the network was the LTU leg. The transmitting radios from the center of town to all the homes were three Ubiquiti LTU Rockets. These radios connected to an Ubiquiti LTU Pro at each house, which provided Internet connectivity to the user.

3.1.2 ZoomTEL Testbed

The second testbed topology was not deployed to an actual location where it would be permanently used, and was instead just tested locally. The topology of what was tested can be seen in Figure 3.2. The primary idea behind this network topology, and what

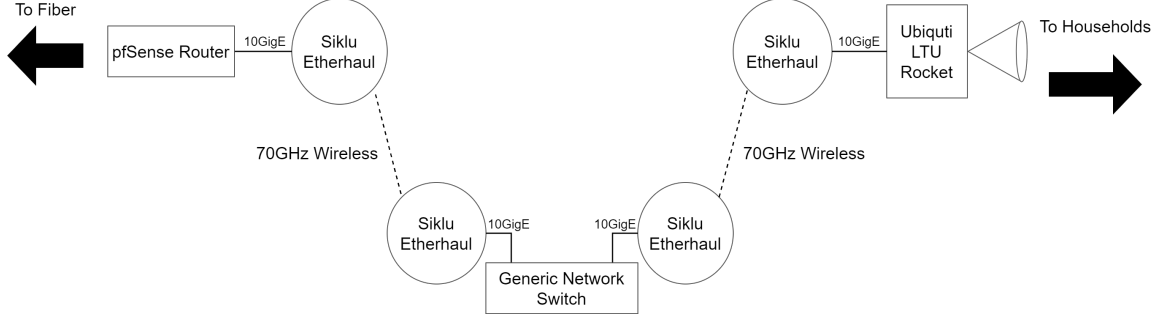


Figure 3.2: The network topology of the experimental testbed built as part of the ZoomTEL project.

makes it differ from the OVERCOME topology, is that the segment shown in Figure 3.2 is intended to be repeated and tiled to create a mesh of backhaul connections. This would allow the pfSense route at the origin of the mesh to make routing determinations based on the quality of links in the network, and could attempt to solve the issue of the delicate nature of mmWave links, especially when concerning weather systems. In this thesis only the single segment shown in Figure 3.2 was implemented for testing, but could easily be expanded in EMANE.

The technologies used in this testbed are similar to the ones in the previous network, but the equipment that implements primarily the mmWave links is different. In this network, the fiber is directly connected to a pfSense router instead of the backhaul, this allows the special routing functionality that was described. The hardware network tested in this topology used a generic Dell OptiPlex desktop as the router. Since the pfSense software is free to use, a commercial solution does not need to be purchased to use it. All the mmWave links were created using the Siklu Kilo Series EtherHaul 1200, four of which were used. The LTU link used the same Ubiquiti LTU Rocket as the radio located at the backhaul, but unlike the OVERCOME project, the LTU Lite radios were used as the customer premises equipment (CPE).

3.2 Creating the Networks in EMANE

In order to implement the mmWave and LTU wireless technologies into EMANE, the rfPipe model was selected. Since the primary concern with the emulation testbed was mimicking the general behavior of the network, it was decided that the rfPipe model would work well enough, and a more complex model was not needed. Table 3.1 outlines the key configuration parameters that were used to create the mmWave links. Table 3.2 outlines the same parameters and their selected values for the LTU waveforms. Since the LTU radios varied between testbeds and the characteristics of the basestation radio and CPE radios were different from each other, values that would most accurately model the average expected behavior were selected. This could cause some variation in the results, but it was deemed acceptable for the proposed use case. Most of these parameters were selected based on the datasheets of both radio platforms and the antennas (integrate or external) that were used. This is information that would be available prior to purchasing hardware to validate a design. The freespace pathloss model was selected as the effects of multipath were not expected to be a primary concern in the testbeds.

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

With the radio plugins configured in the NEMs, two EMANE environments were created for the OVERCOME testbed, and three EMANE environments were created for the

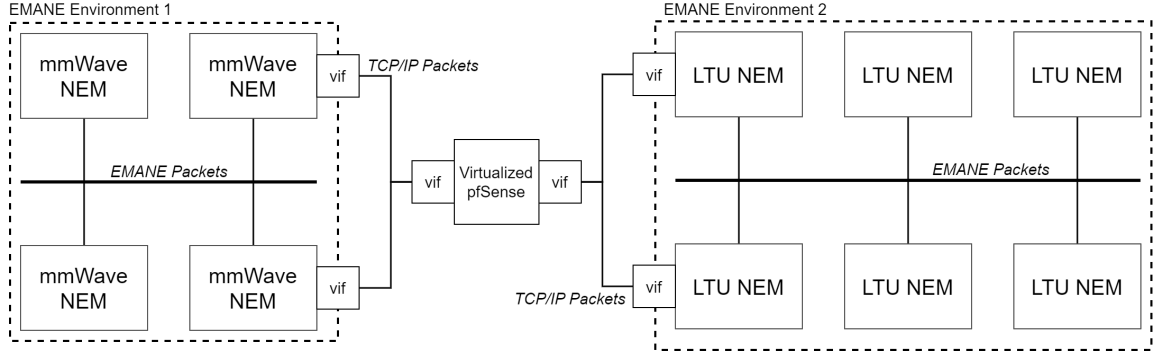


Figure 3.3: The emulation testbed topology corresponding to the OVERCOME project.

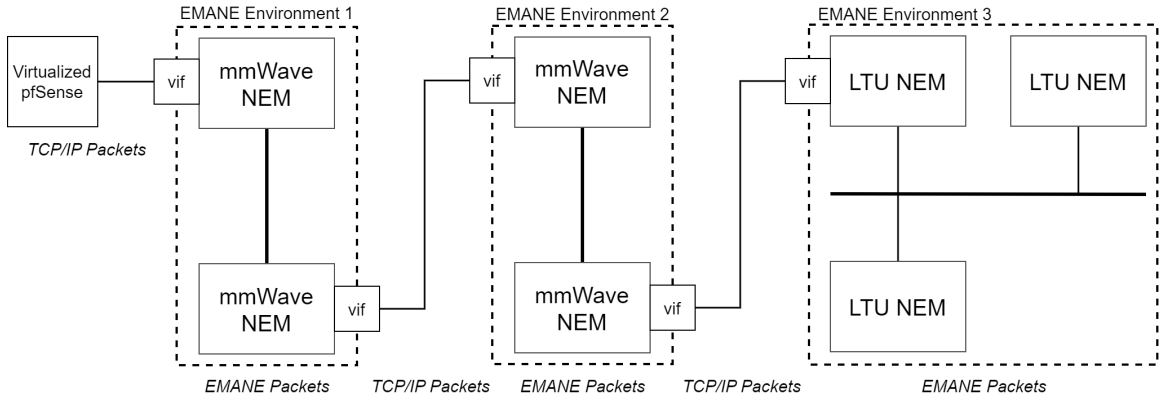


Figure 3.4: The emulation testbed topology corresponding to the ZoomTEL project.

ZoomTEL testbed. These emulation topologies can be seen in Figure 3.3 and Figure 3.4, respectively. It is important to note that the two individual mmWave links in the ZoomTEL topology are separate EMANE instances. These could have been put in the same environment as was the case with the OVERCOME testbed, however it was elected to make them entirely separate to allow for more control over the events that were used to effect the operational environment. If a weather system and its effects on the channel were to be modeled, being able to only impart those effects on a single mmWave link is a very useful ability to have, especially if the testbed was developed further to implement the poor link quality avoidance routing system previously described.

In both testbeds, a pfSense router is virtualized and inserted into the emulation loop. This was initially a difficult process as understanding exactly how the transports needed to be configured to properly pass the TCP/IP traffic to and from the router to emulator

was not clear in the documentation. Several portions of the example configurations use a transport plugin scheme that is later recommended to not be used, causing lots of confusion. Eventually a tutorial [32] and a forum post [33] were found that detailed exactly how to connect a virtual machine to a NEM container. The process primarily relies on creating an additional pair of virtual network interfaces that connect the LXC to the host machine, and having VirtualBox bridge this interface. Since this new interface is not a part of EMANE directly, and therefore not a member of any routing schemes, manual routes needed to be configured so the LXC would properly forward the packets to and from EMANE. This is an example of the code that was used to add these manual routes in the ZoomTEL testbed:

```
#!/bin/bash -
ip route add 10.150.2.0/24 via 10.100.1.2
ip route add 10.100.2.0/24 via 10.100.1.2
ip route add 10.150.3.0/24 via 10.100.1.2
ip route add 10.200.3.0/24 via 10.100.1.2
```

This script would run on startup of the initial mmWave LXC node and add routes to the second mmWave emulation network, the LTU network, and the external TCP/IP networks bridging the EMANE instances.

One of the important factors taken into consideration when virtualizing pfSense, was if the virtualization of the router software would have performance impacts on the network. The expectation was that there would not be any major impacts since pfSense is a very lightweight program and is often virtualized in production environments. For reference, the pfSense router running in production in the OVERCOME project connecting 30 households to the Internet was never recorded at more than a few percent CPU usage. Similarly, the virtual pfSense router never reached above 5% CPU usage and the router never indicated it was dropping packets due to computational overload.

3.3 Hardware Results versus Emulation Results

Having built the emulation testbeds, it was time to use them for testing. Since the hardware testbeds were already in the process of being built, data could be used off the hardware testbed to determine the legitimacy of results produced by EMANE. If the results showed that the key values like data throughput and latency were accurate, EMANE could be used as a development environment, as detailed in Chapter 4.

In order to get the relevant data from EMANE, tools like *iperf3* and *ping* were used for generating and measuring traffic. The *MGENT* utility was also used, and is an open-source tool designed by the Naval Research Lab. It can be used to generate and log a variety of TCP/IP and UDP/IP traffic and can be used to script traffic behaviors to create repeatable experiments. An example of an *iperf3* test can be seen in Figure 3.5 The ZoomTEL hardware

```

sdr@wilab-zoomtel-a:~$ iperf3 -s
Server listening on 5201
Accepted connection from 192.168.2.2, port 44314
[ 5] local 192.168.2.1 port 5201 connected to 192.168.2.2 port 44316
[ ID] Interval      Transfer     Bitrate
[ 5] 0.00-1.00    sec 53.1 MBytes  446 Mbits/sec
[ 5] 1.00-2.00    sec 53.2 MBytes  446 Mbits/sec
[ 5] 2.00-3.00    sec 53.2 MBytes  446 Mbits/sec
[ 5] 3.00-4.00    sec 53.2 MBytes  446 Mbits/sec
[ 5] 4.00-5.00    sec 53.2 MBytes  446 Mbits/sec
[ 5] 5.00-6.00    sec 53.2 MBytes  446 Mbits/sec
[ 5] 6.00-7.00    sec 53.2 MBytes  446 Mbits/sec
[ 5] 7.00-8.00    sec 53.2 MBytes  446 Mbits/sec
[ 5] 8.00-9.00    sec 52.8 MBytes  443 Mbits/sec
[ 5] 9.00-10.00   sec 52.4 MBytes  440 Mbits/sec
[ 5] 10.00-11.00  sec 52.8 MBytes  443 Mbits/sec
[ 5] 11.00-12.00  sec 52.8 MBytes  443 Mbits/sec
[ 5] 12.00-13.00  sec 52.8 MBytes  443 Mbits/sec
[ 5] 13.00-14.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 14.00-15.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 15.00-16.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 16.00-17.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 17.00-18.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 18.00-19.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 19.00-20.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 20.00-21.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 21.00-22.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 22.00-23.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 23.00-24.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 24.00-25.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 25.00-26.00  sec 53.2 MBytes  446 Mbits/sec
[ 5] 26.00-27.00  sec 53.2 MBytes  446 Mbits/sec

sdr@wilab-etherhaul-1:~$ iperf3 -c 192.168.2.1 -bidir -t 60
Connecting to host 192.168.2.1, port 5201
[ 5] local 192.168.2.2 port 44316 connected to 192.168.2.1 port 5201
[ ID] Interval      Transfer     Bitrate    Retr  Cwnd
[ 5] 0.00-1.00    sec 54.6 MBytes  458 Mbits/sec  5    266 KBytes
[ 5] 1.00-2.00    sec 53.6 MBytes  449 Mbits/sec  2    232 KBytes
[ 5] 2.00-3.00    sec 52.9 MBytes  444 Mbits/sec  1    296 KBytes
[ 5] 3.00-4.00    sec 53.6 MBytes  450 Mbits/sec  7    266 KBytes
[ 5] 4.00-5.00    sec 53.0 MBytes  445 Mbits/sec  3    232 KBytes
[ 5] 5.00-6.00    sec 52.9 MBytes  444 Mbits/sec  2    298 KBytes
[ 5] 6.00-7.00    sec 53.7 MBytes  450 Mbits/sec  8    266 KBytes
[ 5] 7.00-8.00    sec 53.0 MBytes  445 Mbits/sec  3    230 KBytes
[ 5] 8.00-9.00    sec 53.0 MBytes  445 Mbits/sec  2    297 KBytes
[ 5] 9.00-10.00   sec 52.4 MBytes  440 Mbits/sec  8    270 KBytes
[ 5] 10.00-11.00  sec 52.4 MBytes  440 Mbits/sec  10   242 KBytes
[ 5] 11.00-12.00  sec 52.7 MBytes  442 Mbits/sec  3    212 KBytes
[ 5] 12.00-13.00  sec 53.0 MBytes  444 Mbits/sec  2    276 KBytes
[ 5] 13.00-14.00  sec 53.1 MBytes  445 Mbits/sec  9    238 KBytes
[ 5] 14.00-15.00  sec 53.0 MBytes  444 Mbits/sec  1    300 KBytes
[ 5] 15.00-16.00  sec 53.7 MBytes  451 Mbits/sec  2    269 KBytes
[ 5] 16.00-17.00  sec 53.0 MBytes  444 Mbits/sec  2    239 KBytes
[ 5] 17.00-18.00  sec 53.0 MBytes  445 Mbits/sec  2    298 KBytes
[ 5] 18.00-19.00  sec 53.6 MBytes  449 Mbits/sec  5    270 KBytes
[ 5] 19.00-20.00  sec 53.1 MBytes  445 Mbits/sec  5    233 KBytes
[ 5] 20.00-21.00  sec 52.9 MBytes  444 Mbits/sec  2    297 KBytes
[ 5] 21.00-22.00  sec 53.6 MBytes  449 Mbits/sec  3    269 KBytes
[ 5] 22.00-23.00  sec 53.1 MBytes  446 Mbits/sec  6    235 KBytes
[ 5] 23.00-24.00  sec 53.0 MBytes  445 Mbits/sec  1    298 KBytes
[ 5] 24.00-25.00  sec 53.6 MBytes  450 Mbits/sec  4    270 KBytes
[ 5] 25.00-26.00  sec 53.0 MBytes  445 Mbits/sec  3    240 KBytes
[ 5] 26.00-27.00  sec 52.9 MBytes  444 Mbits/sec  2    298 KBytes
[ 5] 27.00-28.00  sec 53.0 MBytes  445 Mbits/sec  2    277 KBytes
[ 5] 28.00-29.00  sec 53.0 MBytes  444 Mbits/sec  2    245 KBytes
[ 5] 29.00-30.00  sec 53.0 MBytes  445 Mbits/sec  3    303 KBytes

```

Figure 3.5: The console output of an *iperf3* test used to find the throughput for part of the ZoomTEL EMANE testbed.

testbed also used the *iperf3* utility for testing. The following command was used for running a test on the client:

```
iperf3 -c <server ip address> -bidir -t 60
```

This command sets the destination of the server (the node being transmitted to) with the "-c" flag. The "-bidir" flag indicates the test should be run in bidirectional mode so that both the throughput of the uplink and downlink can be measured. The final flag "-t 60" indicates that the test should run for 60 seconds. This is to ensure a stable enough average is measured since the total throughput can have slight fluctuations.

The OVERCOME testbed utilized several tools for recording data. Throughput and latency tests were measured via the website *speedtest.net*. This tool was used by the engineers at the Internet Service Provider that installed the equipment at the users home, and the aggregate data was found by taking an average of all results. Users that lived closer to the center of town had better results, but this was offset in the average by houses further away. Table 3.3 shows the average results from all three environments for comparison.

Table 3.3: Latency and throughput measurements from the Projects OVERCOME, ZoomTEL, and EMANE testbeds.

	Throughput (Upload)	Throughput (Download)	Total Throughput	Latency
OVERCOME	62 Mbps	275 Mbps	337Mbps	6ms
ZoomTEL	87 Mbps	87 Mbps	174Mbps	5.5ms
EMANE	96.6 Mbps	96.7 Mbps	193.2Mbps	10ms

There are several discrepancies in the data to address. These discrepancies do not necessarily imply the EMANE model is unreliable, but it does mean that further refinement and testing should be conducted to ensure accuracy. Without conducting this further verification, EMANE can not be used on its own for testing as the results can not be fully trusted.

The first point of interest is the difference in total throughput between OVERCOME and EMANE. OVERCOME has a much higher total throughput than EMANE, but this is likely attributed to inaccurate configuration of the CPE devices abstracted in EMANE. The OVERCOME hardware testbed used higher end LTU radios as CPE and the configuration of the LTU model in EMANE was aligned more with the inexpensive LTU units used in the ZoomTEL testbed. This is why the results from ZoomTEL and EMANE are much similar. The uneven speeds in the OVERCOME testbed were a design decision made to allow homes to have higher download speeds. This asymmetric behavior could not be modeled in EMANE as the generic rfPipe model does not support this functionality. The latency differences between EMANE and the hardware testbeds is possibly attributed to the packet completion rate curves for the rfPipe models. These curves may not have been modified enough from the baseline to line up with the actual expected behavior. Another possibility is the delay parameters not being configured properly. The values chosen were rather conservative estimates and may have been too high to properly model the accurate behavior of the hardware wireless signal.

Looking at the throughput and latency of the rfPipe model is not necessarily enough to declare that rfPipe is fully accurate to a hardware model as factors like power and noise calculations should be verified as well. Additionally, since EMANE models are highly configurable, any given configuration would also need to be subjected to some level of scrutiny to ensure it is accurate enough for the intended use case.

3.4 Chapter Summary

This chapter outlines how EMANE was used to create digital models of two separate wireless hybrid network topologies that were designed with the intention of delivering broadband to harder to service areas. The architectures of both testbeds were detailed and the hardware equipment being modeled was outlined. We then explained the parameters in EMANE that were selected in order to mimic the behavior of the hardware radios. Two rfPipe models were created, one for mmWave and one for LTU. After building an extensive enough understanding of the transport models within EMANE, a virtual pfSense router was also brought into the loop to create additional accuracies. The chapter finishes off by showing throughput and latency measurements taken on all testbeds and comparing them. The results are similar enough that the emulator could be used for initial testing in place of the hardware testbed, but the discrepancies show there is room for further refinement of the emulation model. Using a more custom model besides rfPipe could also result in better results, but this new model would need to be vigorously tested to ensure its validity.

Chapter 4

Networking Software Development Environment

The second scenario EMANE was used for was the development of an intelligent resource allocation program. The goal of the network topologies presented in Chapter 3 were to deliver greater amounts of connectivity to underserved rural communities. Introducing new networking hardware is one way of achieving this goal, but another potential for increasing the usability of the Internet is to better allocate resources in an intelligent manner. By using the EMANE testbed developed in the previous chapter, an accurate environment for developing this software can be created. Testing inside an emulated EMANE environment allows the software to still act on an accurate network without needing to be deployed to users. Developing in a production environment could have potentially several issues, such as violating the privacy concerns of users, or diminishing the quality of service a user experiences. This chapter will present the initial creation of a tool to achieve the goal of more intelligently distributing network resources.

4.1 Intelligent Method of Bandwidth Distribution

The original proposal for the software developed in this chapter was to utilize machine learning tools to create a model that would be able to allocate resources in a network with high efficiency. This was deemed to be too complex of a first step and so instead a heuristic approach was decided upon with a special focus on determining if EMANE could make an adequate development environment for this tool. The primary motivation behind the

intelligent router program was the idea that using an identical, static bandwidth cap for each house on the network is not efficient, and even a scheme where certain houses are able to have higher reservations (like most ISPs presently use) would leave too much bandwidth unused. Ideally the only time the network should not be at 100% usage is when there are enough users online to bring it to full capacity. There should not be a situation where a user cannot access resources simply because they are allocated to another user, and that user is not using them.

To facilitate the development of this tool, the OVERCOME EMANE testbed from Chapter 3 was modified to create a better environment to develop in. Figure 4.1 shows the modified environment. The primary difference is the removal of the mmWave model and instead directly connecting the router to the Internet (or other upstream data source). The intelligent router was only concerned with the distribution of network resources from the aggregated central point, and as such the characteristics of the mmWave backhaul were deemed unnecessary to model. The program was eventually deployed to the full OVERCOME EMANE testbed before deployment to hardware, but this was a very minor test to ensure the behavior did not change between virtual environments.

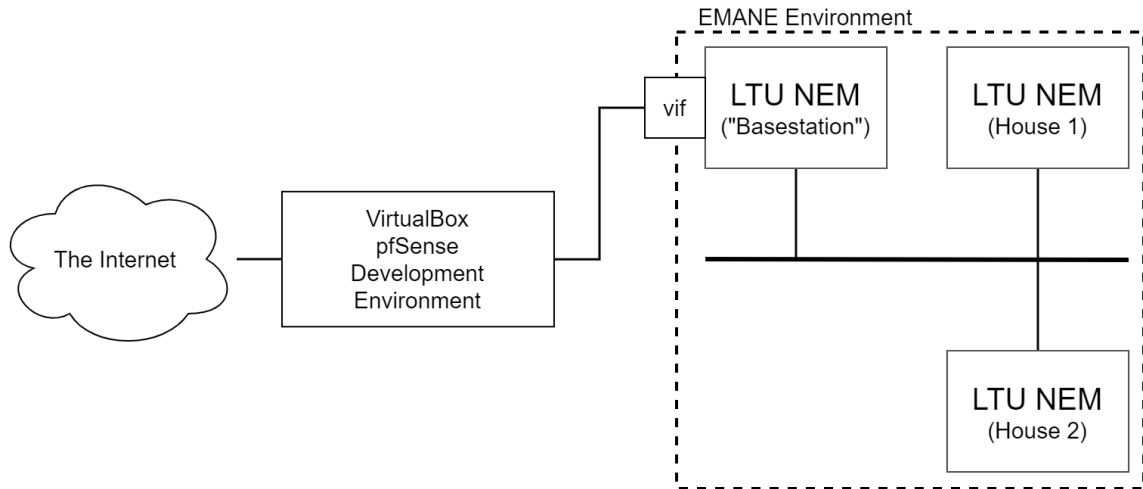


Figure 4.1: The modified OVERCOME EMANE testbed used for development of the intelligent router program. The primary modification is the removal of the mmWave environment.

4.2 Implementing the Software

The algorithm for allocating bandwidth consists of two major stages: classification of the current state of the network, and allocation to modify the state of the network.

4.2.1 Stage 1: Classification

In order to determine how bandwidth should be distributed to users, the program first needed to be aware of the current state of the network. There were a few different tools that were considered to determine the current usage of each host on the network. Because of software was to run on pfSense, this greatly limited the available software. A majority of the usable packages were only the ones made available through the pfSense add-on library. Since the add-on programs and router could not tolerate any possible security vulnerabilities, tools had to be extensively validated before being added to the plugin library. On top of this most of the software made available by pfSense was graphical (as pfSense is effectively a graphical layer for FreeBSD). Needing a tool that would output to the console, or have a redirectable output that could be intercepted by Python *iftop* was selected. This tool takes the average bandwidth traveling through an interface and outputs the values to the terminal. Figure 4.2 shows an example output of this tool. This data was used to aggregate the IP addresses and upload and download bandwidths (in Kbps) for each house present on the network. Figure 4.3 is an example of a log file that shows the data generated by *iftop*. (IP addresses have been censored for user privacy).

The bandwidth data was then used to classify each host into a priority level. In order to not invade the privacy of the users on the network, it was decided that priority levels should not be determined based on the user's specific activity and traffic type. To identify what services each user was using would require examining the user's traffic which is not only computationally inefficient, but also a breach of privacy. In addition to this, our algorithm is not in a position to determine what type of traffic is more important. Identifying the difference between a Zoom meeting and a Netflix stream does not indicate which one should be prioritized and this was not a decision we were in a position to make. Instead, the four created priorities were based on the amount of bandwidth being used. This would give some insight into what the user was doing, and since high bandwidth activities are typically more sensitive to limited datarates, it made sense to prioritize those users. Table 4.1 shows the priority classifications, as well as the thresholds for "high" and "low" bandwidths. The

```

jrmurphy@DAC-VM2 ~
> sudo iftop -i ens18 -c .iftoprc -s 10 -L 10 -t
interface: ens18
IP address is: 10.2.0.9
MAC address is: 00:17:a4:2f:f5:1b
Listening on ens18
# Host name (port/service if enabled)      last 2s    last 10s    last 40s    cumulative
-----
1 *                                          =>         672b         605b         605b         756B
10.2.0.52                                   <=         672b         605b         605b         756B
2 *                                          =>           0b           0b           0b           0B
0.0.0.0                                    <=           0b         262b         262b         328B
3 *                                          =>           0b           0b           0b           0B
10.2.0.17                                   <=           0b         255b         255b         319B
4 *                                          =>           0b         114b         114b         142B
224.0.0.251                                <=           0b           0b           0b           0B
-----
Total send rate:                          672b         718b         718b
Total receive rate:                       672b         1.10Kb         1.10Kb
Total send and receive rate:              1.31Kb         1.80Kb         1.80Kb
-----
Peak rate (sent/received/total):          1.21Kb         3.18Kb         4.39Kb
Cumulative (sent/received/total):         898B          1.37KB         2.25KB
=====

```

Figure 4.2: An example of the output of the *iftop* utility used to measure bandwidth on the OVERCOME network.

```

02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.10; Upload: 308.00Kbps, Download: 34406.40Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.11; Upload: 134.00Kbps, Download: 12595.20Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.12; Upload: 681.00Kbps, Download: 11776.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.13; Upload: 63.30Kbps, Download: 4843.52Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.14; Upload: 22.10Kbps, Download: 1536.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.15; Upload: 208.00Kbps, Download: 3143.68Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.16; Upload: 329.00Kbps, Download: 1177.60Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.17; Upload: 18.20Kbps, Download: 333.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.18; Upload: 45.60Kbps, Download: 82.80Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.19; Upload: 2.17Kbps, Download: 2.18Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.20; Upload: 9.70Kbps, Download: 31.10Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.21; Upload: 27.00Kbps, Download: 18.70Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.22; Upload: 54.10Kbps, Download: 5.76Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.23; Upload: 18.40Kbps, Download: 13.80Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.24; Upload: 0.00Kbps, Download: 5.36Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.25; Upload: 0.00Kbps, Download: 0.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.26; Upload: 1.35Kbps, Download: 928.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.27; Upload: 0.00Kbps, Download: 792.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.28; Upload: 2.38Kbps, Download: 2.80Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.29; Upload: 1.36Kbps, Download: 1.45Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.30; Upload: 416.00Kbps, Download: 416.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.31; Upload: 1.36Kbps, Download: 1.45Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.32; Upload: 880.00Kbps, Download: 1.52Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.33; Upload: 716.00Kbps, Download: 208.00Kbps
02/13/2022 12:36:05 PM - INFO: Host: 10.2.0.34; Upload: 344.00Kbps, Download: 504.00Kbps

```

Figure 4.3: An example of the data output but the classification stage of the intelligent router program. IP addresses have been censored for privacy.

potential was also discussed for adding specific priorities to better control large uploads or downloads, but was never implemented.

Table 4.1: Priority groups for the intelligent router, based on user bandwidth behaviors

	Download Behavior	Upload Behavior
Priority 1	High (>5Mbps)	High (>200Kbps)
Priority 2	High (>5Mbps)	Low (>200Kbps)
Priority 3	Low (<5Mbps)	High (>200Kbps)
Priority 4	Low (<5Mbps)	Low (>200Kbps)

Once the priorities were assigned, the final step was to flag all hosts that needed reallocation. The criterion for receiving reallocation was having a current average usage that was within 5% of the currently set cap. Meeting this criterion would indicate to the allocation stage that the house should receive more bandwidth (if possible). The logic for this reallocation is discussed in the next subsection. Figure 4.4 provides an overview of the classification process. The full source code for the classification stage of the router can be found in Appendix B.

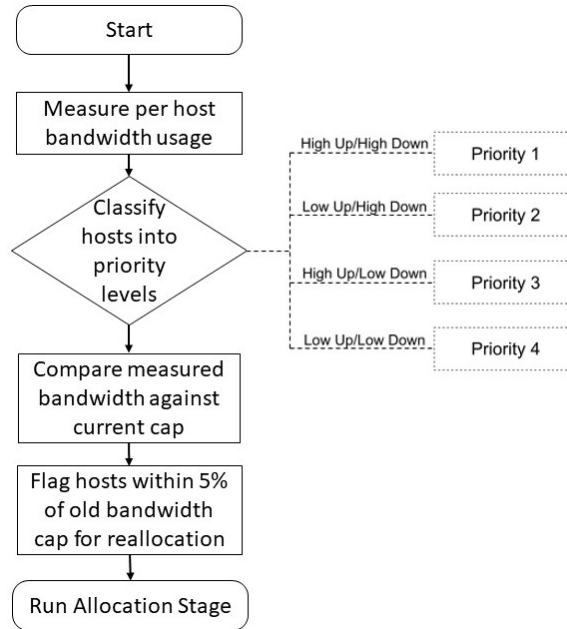


Figure 4.4: An overview of the algorithm that operates to classify each host on the network as a part of the intelligent router program.

4.2.2 Stage 2: Allocation

Once the classification stage was complete, the allocation stage would run. This stage read the data created by the previous stage and acted on it to impact the network and change the amount of bandwidth an individual house was able to use. The primary decisions made while designing the allocation stage related to the safety measures put in place to ensure no user was allowed too much or too little bandwidth. Since the algorithm was set up to continually take and give bandwidth, checks needed to be put in place to ensure a single house was not able to accumulate all the bandwidth, starving the rest of the hosts. After evaluating the average usages for the network, the minimum value a house could have was set to 7.5Mbps. This value was a rather conservative estimate and in all likelihood could have been set lower, however it did not impact the efficiency of the algorithm.

Once measures were put in place to protect the network, the method with which homes were limited had to be decided upon. Since the algorithm implementation was running alongside pfSense, we had to be careful that the method we used to set limits would not be overridden by the primary router software. To ensure this, we used pfSense's Limiters, a group of settings that could be assigned to a firewall rule to ensure a maximum throughput for any traffic caught by the rule. This method worked well because each house could be assigned a firewall rule based on its IP address. Any traffic destined to that IP address would pass through the limiter. Additionally, because these settings were stored in the main configuration XML file, the Python script running the algorithm could easily edit and update the file, instead of trying to fight against the default configuration.

With these design decisions out of the way, the steps that needed to be taken to allocate traffic were obvious. The first step was to look at the total amount of bandwidth allocated and determine if the network was at saturation. This was done by keeping a record of the sum of all allocations that was then compared against the known network total. If bandwidth was available, the algorithm would proceed to increase every flagged host's bandwidth cap by 5Mbps. If there was no extra bandwidth available, lower priority hosts that did not require as high limits would have their caps lowered, and the additional bandwidth could be redistributed. Once all flagged hosts either had their caps raised, or were deemed unable to receive more bandwidth, the program would wait six seconds before returning to the classification stage. This waiting period is primarily to account for inefficiencies in the pfSense system. Since pfSense was directly being used to change bandwidth caps, a small waiting

period was required to allow the values to propagate through the system. The full behavior of the allocation stage can be seen below in Figure 4.5, and the full source code for the allocation stage of the router can be found in Appendix B.

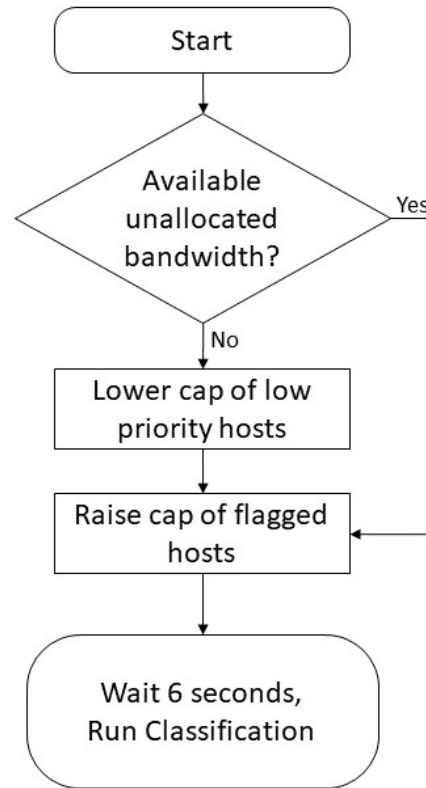


Figure 4.5: An overview of the algorithm that operates to allocate bandwidth to each host on the network as a part of the intelligent router program.

4.3 Effectiveness of the Program

Whether or not EMANE makes an appropriate environment to develop a tool like this is difficult to answer from the observed data and results. One of the primary benefits of developing in EMANE is that it is a closed environment that protects the program and developer from errors effecting the quality of a network. If the tool were to be developed in production, any small configuration error could easily result in taking down a network many people rely on. Conversely, developing on a generic computer, or a router connected to only a very limited number of hardware devices does not provide the traffic or tools necessary

to test the behavior. By operating in EMANE, any number of "houses" can be connected downstream from the router, the amount of traffic being created at once and which hosts the traffic comes from is controllable, and the attributes of the communication medium can be modified.

This makes development very convenient, however, it is not a catch-all solution. One of the major problems discovered during development is that the generated test traffic did not resemble the behavior of the real user traffic enough. Traffic generated by a test tool like *MGEN* is typically not dynamic enough to mirror the behavior of an individual, let alone an entire household. Extend this to covering the entire network, and generating test traffic that is accurate for five or ten households is a difficult task.

One solution attempted during the development of the tool, was to use a packet capture (in the form of a PCAP file) to record the traffic of consenting users and play it across the network. The problem that was found with this solution was that different demographics will have drastically different traffic usage behavior. The Internet usage of a New England college student does not necessarily match that of an adult in a rural area like Missouri. This became a problem during testing as the algorithm was reacting slower to bandwidth needs than in testing, as the high usages were more sporadic so the averaging approach at classification tended to trend lower than required.

The other major result from the testing of the router, was the discovery that effective testing of a program like this requires an ideal environment to test in. The OVERCOME hardware network was not constrained enough due to the small number of houses connected. An attempt was made to constrain the network during testing, but was overall unsuccessful. The hardware testing schedule proceeded as follows:

- **Week 1:** No artificial bandwidth limits, no intelligent program.
- **Week 2:** All homes restricted to 25Mbps download in an attempt to create a lower max network capacity.
- **Week 3:** Homes were assigned dynamic restrictions based on the algorithm.

Figure 4.6 shows the total usage of the network during each classification iteration for the entire week 2 and 3 testing period. As can be seen by the blue traffic data, at no point during the two-week period was the usage in the network even at 50% capacity. This caused the intelligent algorithm to never have to remove bandwidth from a user to give to another

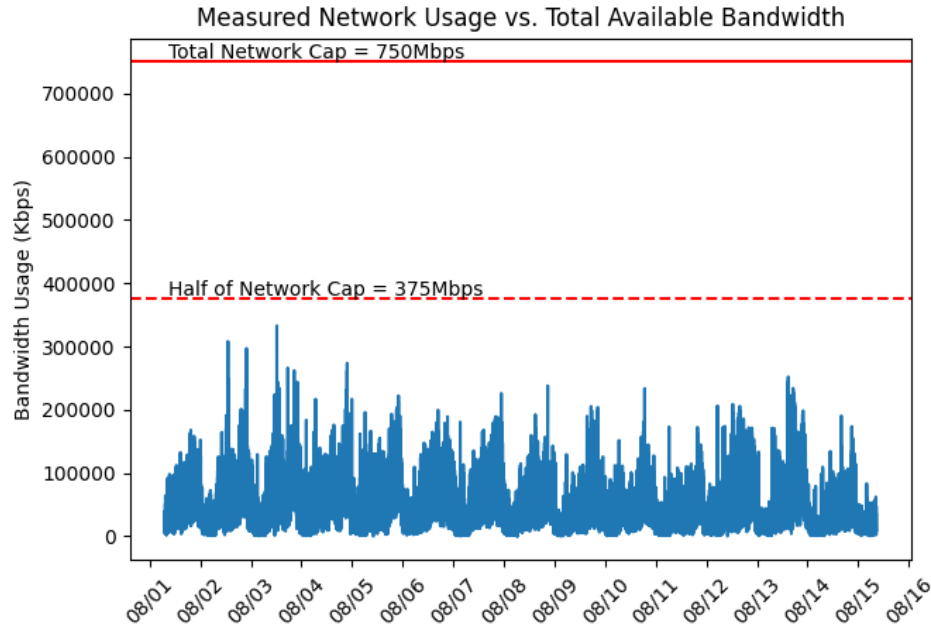


Figure 4.6: The total network usage for thirty houses in the Project OVERCOME testbed during weeks 2 and 3 of the intelligent router test.

user, eliminating the point of the program. The artificial cap on the network also could not be lowered further, because the individual households would have such a low base restriction that users would be barely able to use the Internet during the control period. With the project near completion, it was determined that another test could not be run, and the algorithm would have to remain unfinished as future work.

One positive result from the test, however, was data confirming that a stratification of usage existed on the network. By confirming that there is a distribution of users that use a lot of bandwidth and users that use little, it supports the core concept of the algorithm that a flat distribution cap on a network, while fair, is not the most efficient use of resources. Figure 4.7 shows an example of two houses on the network. The top house with much higher usage, and the bottom house with less usage.

4.4 Chapter Summary

This chapter outlined the process for designing a piece of networking software while using EMANE as the network environment for development. More specifically, a program that sought to heuristically allocate network resources in an intelligent manner was outlined. The

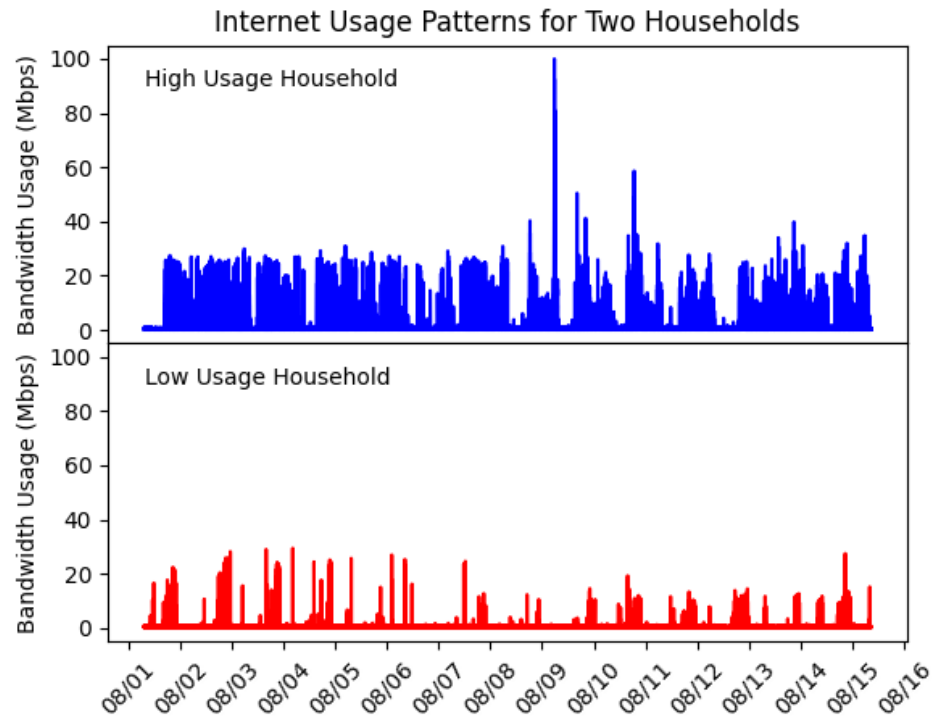


Figure 4.7: The total network usage for thirty houses in the Project OVERCOME testbed during weeks 2 and 3 of the intelligent router test.

specifics of the environment the tool was developed in were highlighted and the behavior of the algorithm explained. The chapter finalizes by outlining lessons learned from the development and testing of the tool, with the main takeaway being emulation environments need to also validate the behavior of the test traffic, in addition to the models provided and used.

Chapter 5

Dynamic Robot Swarm Networks

This chapter presents the third and final use case for EMANE that was explored in this thesis. The primary work that will be covered relates to extending the robot swarm simulator ARGoS [34], to introduce emulated communication models. The network outlined in this chapter consists of a swarm of 25 simulated flying robots that are tasked with identifying wildfire in an area. By operating in the context of a wildfire, several environmental factors are introduced that must be considered. Primarily, the conditions this network is present in are not prime for wireless communications as smoke or other obstacles may exist, and well-deployed, centralized infrastructure is unlikely to be present. To this end, communications must happen in a distributed manner across the swarm mesh, and traffic must be relayed to reach a command center. These conditions are the ideal operating parameters for a mobile ad-hoc network (MANET) configuration, which is EMANE's specialty. By using EMANE and ARGoS together, this unique network behavior can be appropriately captured.

5.1 Extending Existing Software

The robot swarm simulator, ARGoS, is a physics based simulator designed with the intent filling the gap for large-scale heterogeneous robot swarms [34]. The tool models the swarm and controls the environment it operates in, modeling the movement, physics, and information transfer of the individual robots. One of the key attributes about ARGoS that is essential to understand, is the manner in which ARGoS performs its simulation. ARGoS will simulate the functionality of the swarm in specified time chunks. This is different

from EMANE which operates in real-time, and needing to account for these fundamental differences influences the design process of integration. There are several reasons why it is beneficial to integrate the two tools. For one, EMANE on its own does not handle the mobility of nodes, the traffic that travels the network, or the management of an environment. These are all things that must be set up externally, and as such having ARGoS handle them is a natural fit. In the other direction, having EMANE handle communication for ARGoS can allow more accurate modeling of channel effects. ARGoS has basic "medium" plugins (the type of plugin responsible for communications), but these can be rather simple. One such model, simply determines if two robots are within a set range, and have line of sight. If these conditions are true then the two robots are free to exchange information with no delay or maximum data rate. Another benefit to using EMANE is the ability to use the native implementation of the B.A.T.M.A.N. routing protocol. By deploying this protocol on EMANE, it can be further tested and developed in a proper swarm environment.

5.2 Integrating the Software

This section will detail the integration of EMANE and ARGoS and how the two software operated together. The following section will shed light on several of the design decisions that were made and explain the rationale behind them.

ARGoS and EMANE both ran on the same machine for the purposes of their integration. EMANE also had an additional program that acted as an intermediary between it and ARGoS, as EMANE is not a single process and needed something to manage it. Each process was also completely separate from the other, neither process was a child. This is important to understand as it dictated how the two processes attached to each other. If one process was a child of the other, they would naturally be connected. Instead, part of the initialization processing was the two processes connecting. ARGoS began the process by opening a shared memory location with a name known by both ARGoS and EMANE. This shared memory contained metadata pertaining to the simulation including the process ID (PID) of ARGoS, the number of robots in the experiment, and the timescale being used. Table 5.1 details the exact content of the shared metadata. After populating this data, ARGoS put itself to sleep and waited for EMANE to indicate it was ready.

EMANE started up during this time and waited until it could find the shared memory structure and to get ARGOS's PID. With this PID, EMANE was able to directly communi-

Table 5.1: Contents of the shared memory metadata file. Includes which processes are responsible for what data

Provided By	Type	Data
ARGoS	uint16_t	Number of Robots
ARGoS	uint16_t	Number of Communication Robots
ARGoS	double	ARGoS Timescale
ARGoS	pid_t	ARGoS Process ID
EMANE	pid_t	EMANE Process ID

cate with ARGoS. EMANE then set up the remaining two shared memory locations, and set up its internal data structures. The second shared memory structure was used by ARGoS to deliver information about the location and pose of robots to EMANE. This was essential for ensuring both programs had the same representation of the virtual environment. Table 5.2 outlines the exact contents of this memory block. The third shared memory structure was

Table 5.2: Contents of the shared memory robot pose file. Includes which processes are responsible for what data

Provided By	Type	Data
ARGoS	uint16_t	Robot ID
ARGoS	double_t	Robot Latitude
ARGoS	double	Robot Longitude
ARGoS	double	Robot Altitude

used by both ARGoS and EMANE to deliver relevant data to performing communications. ARGoS used the block to make requests of EMANE and EMANE used it to response with the requested information. Table 5.3 outlines the exact contents of this memory block.

Table 5.3: Contents of the shared memory robot communications file. Includes which processes are responsible for what data

Provided By	Type	Data
ARGoS	uint16_t	Transmitting Robot ID
ARGoS	uint16_t	Receiving Robot ID
ARGoS	uint8_t	Message Pointer
ARGoS	uint32_t	Message Size
EMANE	uint32_t	Amount of data transmitted

Having set up all the relevant structures used to communicate, the processes were ready to proceed with actual simulation. This began when EMANE woke up ARGoS for the first time. The two processes controlled each other via POSIX signals, specifically the *SIGSTOP*

and *SIGCONT* signals. These signals indicated to the operating system if a process should be put to sleep or woken from sleep. When a process was done with its turn, it signaled the other with a *SIGCONT* and then raised its own *SIGSTOP* to indicate to the operating system to put it should be put to sleep. The main loop then began between the programs. ARGoS first took its turn, moving robots, sensing the environment, and making determinations about information sharing for 100ms of simulation time. Once it finished the period, it updated the relevant information across the shared memory segments and initiated EMANE. The EMANE interface script woke up and updated its internal store of information. Any new location data was sent into the emulator via location events so that the corresponding NEMs would move into the appropriate location and update their communication links. EMANE then began the process of emulating communications. This was done by measuring the throughput to adjacent nodes, instead of performing actual data transfers. The reasoning for this is explained in the following section. Once EMANE collected all the required information from its throughput tests, it populated that information back into the shared memory blocks and woke ARGoS, starting the process over again. This then continued until ARGoS finished its current experiment, at which point it either reset, or terminated. In the case of termination, ARGoS signals EMANE to also shutdown. If ARGoS is immediately starting another experiment, EMANE is not informed as the transition between experiments is entirely transparent to EMANE. Figure 5.1 shows the layout of all the pieces of software required to connect EMANE and ARGoS. Appendix C contains the full source code of the EMANE side of the integration process.

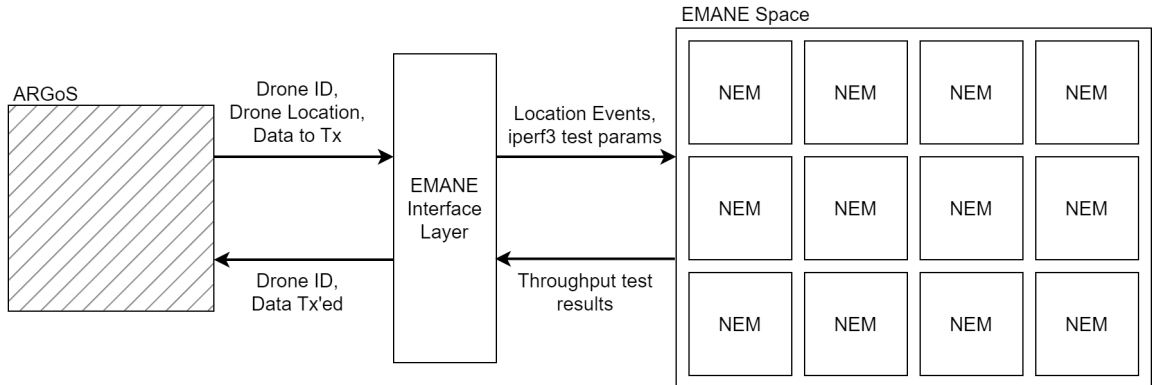


Figure 5.1: The topology of the ARGoS-EMANE integration system. All the major systems as well as the interconnections between them are displayed.

5.3 Integration Design Decisions

Several design decisions were made during the development of the above process with the goal of maximizing the performance of the interface, minimizing the latency between each process running, and ensuring the system remained stable. The two biggest decisions made were the decision to use shared memory for information sharing and the decision to abstract the data being sent through EMANE. These decisions ensured that the interface would still be accurate, without introducing unnecessary complexities that would further slow down the system.

It was decided that one or more shared memory segments should be used to pass information between processes. Transferring data over the network was deemed needlessly slow and complex. The time taken to output data from the tool, package it appropriately, send it over the network, unpack it, and import it would have taken an order of time longer than the actual time needed to generate and consume the data being sent. Since networking was not to be used, both processes needed to run on the same machine and have access to the same hardware. For the same reason that using networking would be too slow, performing file I/O to read and write data to a file would have also been too slow. Since both processes are capable of accessing memory incredibly quick, using the operating system to grant ARGoS and EMANE access to the same location in system memory was the logical solution.

The second major design decision, was to not provide EMANE with the exact data payloads ARGoS was sending between robots. ARGoS, and the algorithm running on the robots, was not concerned with any form of transmission control or retransmission upon failed delivery of data. This means that EMANE did not need to worry about what data successfully made it to the receiver, just how much of it arrived. By avoiding handing off the entire data payload to EMANE, time could be saved and a layer of complexity was removed. EMANE instead can just characterize the transmission, and inform ARGoS on what amounts of data can be given to each robot in a certain timeframe. This does not compromise the validity of the channel effects EMANE imparts on transmission, as throughput and latency were the primary two metrics being modeled anyway (as witnessed in Chapter 3).

One of the major issues during development that had to be addressed was the difference in programming languages used. ARGoS, and therefore its portion of the interface code, was

written in C++. The EMANE interface manager, was written in Python. Since data was being accessed directly from memory, it was important that the exact size of the data was accurately read. C++ provides the user with data types that are not present in Python, with most data types in Python taking up larger amounts of space than the closest equivalent in C/C++. Even when using a library like *ctypes*, meant to introduce more "C-like" types, the problem was still present. The solution ended up being to serialize and deserialize the data instead of mapping the memory directly to a structure, like was done in C++. The following code snippet uses a format string to interpret the exact number of bytes each variable should be for serialization and serialization

```
__struct_format: str = 'HHdIIddd'

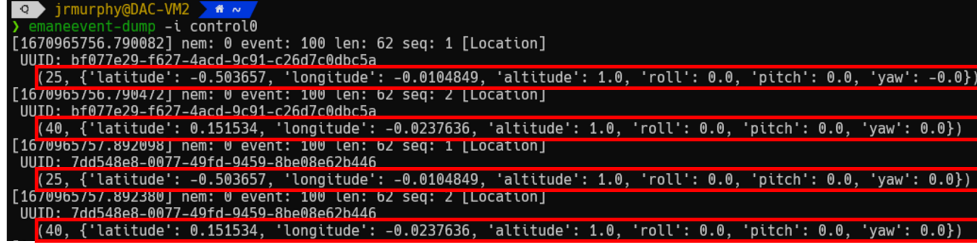
def unpack(self, shm):
    self.num_drone, self.num_comms, self.deltaT, self.argos_pid, self.emane_pid,
    ↪ self.gw_lat, self.gw_lon, self.gw_alt \
        = struct.unpack(self.__struct_format, shm.buf.tobytes())
```

This solution worked well to solve the problem, and while it may have been slower than the process used for memory access in C++, it was still much faster than EMANE takes and did not affect overall run time. Further testing can be performed to determine if the interface script would have an improvement on performance if written entire in C++.

5.4 Integration Results

The basic ability for ARGoS and EMANE to pass information back and forth is present, with ARGoS delivering robot locations and EMANE delivering communications metrics, but several factors remain to be implemented or improved upon. Figure 5.2 shows a printout of EMANE's event channel. By using the *emaneevent-dump* utility, any events sent through EMANE can be seen. As witnessed in Figure 5.2, EMANE received location events over time as ARGoS updated the manager script with new locations. It can be seen from the initial numbers in square brackets that these events were only firing about every one second. Since these events should be firing for every 100ms of simulation time, we see that simulating 100ms of time took ARGoS and EMANE a combined ten times longer. This behavior was expected from the start, but the extent of how bad it would be had been unknown.

EMANE runs in real-time and therefore at a minimum would take 100ms to create



```

jrmurphy@DAC-VM2 ~
> emanevent-dump -l control0
[1670965756.790082] nem: 0 event: 100 len: 62 seq: 1 [Location]
  UUID: bf077e29-f627-4acd-9c91-c26d7c0dbc5a
  (25, {'latitude': -0.503657, 'longitude': -0.0104849, 'altitude': 1.0, 'roll': 0.0, 'pitch': 0.0, 'yaw': -0.0})
[1670965756.790472] nem: 0 event: 100 len: 62 seq: 2 [Location]
  UUID: bf077e29-f627-4acd-9c91-c26d7c0dbc5a
  (40, {'latitude': 0.151534, 'longitude': -0.0237636, 'altitude': 1.0, 'roll': 0.0, 'pitch': 0.0, 'yaw': 0.0})
[1670965757.892098] nem: 0 event: 100 len: 62 seq: 1 [Location]
  UUID: 7dd548e8-0077-49fd-9459-8be08e67b446
  (25, {'latitude': -0.503657, 'longitude': -0.0104849, 'altitude': 1.0, 'roll': 0.0, 'pitch': 0.0, 'yaw': 0.0})
[1670965757.892380] nem: 0 event: 100 len: 62 seq: 2 [Location]
  UUID: 7dd548e8-0077-49fd-9459-8be08e67b446
  (40, {'latitude': 0.151534, 'longitude': -0.0237636, 'altitude': 1.0, 'roll': 0.0, 'pitch': 0.0, 'yaw': 0.0})

```

Figure 5.2: A dump of several events that were sent over the EMANE event channel. These location events show that ARGoS is delivering location data to the interface script, which is subsequently delivering location events to EMANE.

100ms of data. However, because the network needs time to reconfigure after locations are updated, and the throughput tests need time to record data, it is much more likely that EMANE takes anywhere from two to five times as long to generate that data. The ten times longer observed above is on the extreme side and was recorded before optimizations were started, however additional optimizations would be needed to bring that number down further, if possible. As a final point to understand the extent of the slowdown imparted on ARGoS by combining it with a real-time emulator, ARGoS running in isolation using an internal communication medium plugin only takes about four seconds to run a 10-minute experiment. With the best case scenario of EMANE adding 200ms of time per 100ms emulation block, the ten-minute experiment would now take twenty minutes. This length of emulation may be an acceptable trade off for the features EMANE brings to the testbed, but is an essential consideration that must be made.

5.5 Chapter Summary

This chapter focused on introducing additional tools to EMANE to enable EMANE to more dynamically model certain networks. By interfacing with the ARGoS robot simulation tool, both tools receive enhancements in the forms of more accurate modeling of all parts of a robot swarm network. The primary results found were that the software are able to communicate through the designed methodology, but the real-time nature of EMANE slows down ARGoS drastically. At the time of writing the interface is in the earlier stages of development with only the basic functionality presented here implemented. The work is being continued in a hope to further optimize the process and deliver additional functionality to both tools.

Chapter 6

Conclusion

6.1 Evaluating Network Emulation

This thesis began by proposing network emulation as a solution to the difficulties present in developing and testing communication networks. Chapter 2 introduced several issues that often occur in network testing and development, including the expensive nature of hardware testbeds, the difficulties in creating an accurate enough testbed, and the time required to test multiple network topologies and configurations. The question now remains if emulation is able to address these issues, and the remainder of this section will attempt to answer this.

Firstly, a few limitations with EMANE and network emulation concepts were discovered in testing that must be considered when determining if emulation is an appropriate tool. The first limitation relates to the performance of emulated networks. All the networks emulated in this thesis were able to match the expected network performance. This was primarily due to the careful configuration of the tool. Every design decision was made while considering the performance impacts they would cause, and even with these considerations, initial configurations had instability. These instabilities are caused by the computational load generated when all packets in a network have to be handled by a single kernel, a problem unique to emulation. The system was eventually made stable by ensuring enough computational resources were available for the task. Still, in bigger testbeds, a single server may not be sufficient, and a distributed approach to emulation may need to be considered. The other primary limitation found with emulation and EMANE was the initial difficulty with using the tool. Emulation was suggested as an easier alternative to other testing methods, and it still could be easier provided enough documentation and tutorials are

created. The difficulty with understanding how to use EMANE was specifically attributed to the tool’s small user base. As more researchers use the tool, it becomes easier to find help. The same can be said for the difficulty in using network simulation and networking hardware. Every tool will have an initial period of difficulty, especially if adequate documentation does not exist, and for this reason, the difficulties experienced with EMANE should not invalidate it.

Despite these limitations, we still found emulation works as a good tool for testing networks. Emulation primarily lowers the cost of testing, especially when using an open-source tool like EMANE that costs nothing to use. For comparison, the hardware alone in the testbed built in Chapter 3 cost over \$15,000 USD, excluding costs for labor or permits needed to deploy the network. This significant difference in cost is a strong argument for simulation and emulation. Another issue proposed was the difficulty in creating an accurate testbed. Our experiments showed that in addition to being flexible, EMANE is also able to model several factors of a communications network accurately. In addition to writing models that provide accuracy, hardware can be interfaced with the tool (as shown in Chapter 4) or other software tools can be integrated to provide accuracy for externally controlled factors (as seen in Chapter 5). This desire for an accurate representation of the physical channel provides emulation with an advantage over simulation, as being able to use hardware for specific segments can ensure perfect accuracy. As long as the tool is used with care to ensure accuracy (as is the case with most testing tools), network emulation can provide an ideal testing environment for communication networks.

6.2 Research Outcomes

Overall, this project was able to achieve its primary goal of evaluating and understanding the EMANE tool. Several scenarios were presented that EMANE was shown to be successful at operating under. The first use-case of rural broadband testing, showed that the basic rfPipe radio model can be configured to match the characteristics of a hardware testbed. This scenario does carry the warning that just because the characteristics of the network are the same, not every model in EMANE is validated, and care must be taken to ensure the emulator is configured right. The rfPipe model is simple enough that the throughput and latency matching is enough to use it in basic testing, but advance models mimicking more complex behaviors must be tested further. The second use-case shows that EMANE

provides a good environment for testing and developing networking software. Care must be taken when selecting and modeling test traffic, but EMANE provides many avenues for creating and using test data. The third use case shows that EMANE is able to work with other tools to create a more complete system, with the timescale differences between software types identified as the major obstacle to be considered.

6.3 Future Work

While each stage of this thesis had successes, there are still several pieces of work that remain uncompleted. The following list outlines this future work:

- Conducting extensive validation of the models included within EMANE. EMANE is not as widely used as many other network testing tools, and as such has been studied less regarding its validity. If more advanced models are to be used in EMANE, it should be ensured they are accurate enough for the advanced use cases.
- Creating additional wireless models for EMANE. By default, EMANE only has four radio models accessible, with one being the generic model used in this thesis and another being used for testing of EMANE itself. If EMANE is to be more widely used, it needs to be able to model more modern waveforms and technologies.
- Additional testing and development of the intelligent router software. Due to the issues described in Chapter 2 only one initial test was conducted with non-ideal conditions. Further testing should be conducted to evaluate the tool in ideal conditions
- Further integration and optimizations between EMANE and ARGoS. Finding ways to reduce the time EMANE adds onto the total simulation, while also adding functionality to allow for modeling of more communication factors like latency or advanced routing techniques would allow the integration to be even more useful.

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] E. A. Vogels, "Some digital divides persist between rural, urban and suburban america," 2021. [Online]. Available: <https://pewrsr.ch/3k5aU6J>
- [3] "Internet/broadband fact sheet," 2021. [Online]. Available: <https://www.pewresearch.org/internet/fact-sheet/internet-broadband/>
- [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.
- [32] S. Galgano, “letce2-tutorial: Experiment 3,” 2021. [Online]. Available: <https://github.com/adjacentlink/letce2-tutorial/tree/master/exp-03>
- [33] —, “How do i connect ”emane0” from one virtual machine (vm) to ”emane0” on another virtual machine (vm)?” 2019. [Online]. Available: <https://github.com/adjacentlink/emane/issues/102>
- [34] C. Pinciroli, V. Trianni, R. O’Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, “ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems,” *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.

Appendix A

Installation of EMANE

1 EMANE + Addons v1.3.3:

2

3 Step 1: Install EMANE Bundle

4 \$ cd EMANE_1.3.3-1_debs

5 \$ sudo dpkg -i *.deb

6 \$ sudo apt install -f

7

8 Step 2: Verify

9 \$ emane -v

10

11 OLSRD v0.9.8

12 Step 1: Dependencies

13 \$ sudo apt install bison flex

14

15 Step 2: Download Release

16 \$ wget

↪ <https://github.com/OLSR/olsrd/wiki/files/0.9.8/olsrd-0.9.8.tar.gz>

17 \$ tar xvf olsrd-0.9.8.tar.gz

18

19 Step 3: Make and Install

20 \$ cd olsrd-0.9.8


```
21         $ make
22         $ sudo make install
23
24     Step 4: Verify
25         $ olsrd -v (NOTE: Might need to add install location
26         ↪      [/usr/local/sbin] to PATH)
27
28     OLSRD Plugins
29
30     txtinfo:
31         $ cd lib/txtinfo
32         $ make
33         $ sudo make install
34
35     bmf:
36         $ cd ../bmf
37         $ make
38         $ sudo make install
39
40     iPerf v3.7
41         $ sudo apt install iperf3
42
43     gpsd + gpsd-clients v3.20
44         $ sudo apt install gpsd gpsd-clients
45
46     BATMAN:
47         $ sudo apt install batctl
48         $ sudo modprobe batman-adv
```

Appendix B

Intelligent Router Source Code

B.1 Control Script

```

1  #!/usr/local/bin/python3.8
2  import time
3  import datetime
4  import logging
5
6  import classify
7  import allocate
8
9  if __name__ == '__main__':
10
11     date = datetime.date.today()
12     logFileName = "./Logs/{today}.log".format(today=date)
13
14     # Setup logging
15     logging.basicConfig(filename=logFileName, filemode='a', level=logging.DEBUG,
16                         format='%(asctime)s - %(levelname)s: %(message)s',
17                         ↪ datefmt='%m/%d/%Y %I:%M:%S %p')
18
19     nextStart = time.monotonic()
20     delta = 7.5 # 7.5 seconds between runs (time to perform all data collection
21               ↪ and processing)

```

```

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
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
25 ↪ bandwidth)
26 data_list = []
27
28 for i in range(len(top_list)):
29     if re.search('^ {1,3}[0-9]', top_list[i]):
30         data_list.append(top_list[i])
31         data_list.append(top_list[i + 1])
32         i + 1
33
34 # Count = 2 * number of hosts
35 # Two lines per host (one upload, one download)
36 count = len(data_list)
37
38 # If list is empty, no data to work on
39 if count < 2:
40     return -1
41
42 # Dictionary to hold host information, in-coming, and out-going traffic
43 global host_dict
44 global host_list
45
46 host_dict = {}
47 host_list = []
48
49 #
50 # For each host upload/download pair, extract information into format below
51 #
52 # Host / Up Rate / Down Rate
53 # <ip_addr> / Mbps / Mbps
54 # <ip_addr> / Mbps / Mbps
55 # "" / "" / ""
56 for i in range(int(count / 2)):
57     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
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,
102             ↪ Priority: %d", ip, bandwidth[0], bandwidth[1], prio)
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"))
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',
137                         ↪ datefmt='%m/%d/%Y %I:%M:%S %p')
138
139     main()

```

B.3 Allocation

```

1  #!/usr/local/bin/python3.8
2  import csv
3  import datetime
4  import logging
5  import os
6  import xml.etree.ElementTree as ET
7
8
9  # Standard amount by which bandwidth should be raised or lowered (Kbps)
10 incrementAmount = 2500
11 decrementAmount = 1000
12
13 # Individual Host max BW and min BW (Kbps)
14 minHost = 7500
15 maxHost = 100000
16
17 # Max bandwidth (Kbps)
18 maxNetworkBandwidth = 25000 * 30
19
20 configFile = 'config.xml'      # pfSense XML setting file (FIXME: Restore this
21 ↪ path to '/conf/config.xml')
22 classifyFile = 'classData.csv' # classify script output file
23

```

```

24  def readClassifyData():
25
26      # Read data from classification script
27      # Dictionary {Key: Value} where Key is <ip address> and Value is (Upload,
        ↪ Download)
28      classifyData = {}
29
30      with open(classifyFile) as csvFile:
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)
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]) -
70                                     ↪ decrementAmount)
71                 logging.info(" ALLOCATE -- Host: %s; Decrease cap, New Cap: %s",
72                             ↪ key, newBWList[key])
73                 continue
74
75         downloadVal = classifyData[key][1]
76
77         if downloadVal > (int(currentAllocation[key]) * 0.95) and
78             ↪ int(currentAllocation[key]) < maxHost:
79             newBWList[key] = str(int(currentAllocation[key]) + incrementAmount)
80             logging.info(" ALLOCATE -- Host: %s; Increase cap, New Cap: %s", key,
81                         ↪ newBWList[key])
82         elif downloadVal < (int(currentAllocation[key]) * 0.50) and
83             ↪ int(currentAllocation[key]) > minHost:
84             newBWList[key] = str(int(currentAllocation[key]) - decrementAmount)
85             logging.info(" ALLOCATE -- Host: %s; Decrease cap, New Cap: %s", key,
86                         ↪ newBWList[key])
87         else:
88             newBWList[key] = currentAllocation[key]
89             logging.info(" ALLOCATE -- Host: %s; No change, New Cap: %s", key,
90                         ↪ newBWList[key])
91
92     return newBWList

```

```

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')
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',
147                             ↪ datefmt='%m/%d/%Y %I:%M:%S %p')
148
149     main({})

```

Appendix C

ARGoS-EMANE Interface Source Code

C.1 Main Code

```
1  #!/usr/bin/env python3
2
3  import os
4  import signal
5  import struct
6  import sys
7  import time
8  from multiprocessing import shared_memory
9
10 from emane.events import EventService, LocationEvent
11
12 from libs.structs import *
13 from libs.drone import EMANEDrone
14
15 META_FORM = 'HHdIIddd'
16 META_SIZE = struct.calcsize(META_FORM)
17 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
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
97      ↳ to start with more nodes than ARGoS)
98      # EMANE can not create new nodes during runtime
99      if(prev_drone < sys_meta.num_drone):
100          shm_pose.close()
101          shm_pose = shared_memory.SharedMemory(name="argos_emane_pose",
102          ↳ create=False)
103          robotpose = [RobotPose() for i in range(sys_meta.num_drone)]
104
105      for i in range(sys_meta.num_drone):
106          buf = shm_pose.buf[i*POSE_SIZE:(i+1)*POSE_SIZE]
107          robotpose[i].unpack(buf)
108          drone_nodes[i].id = robotpose[i].id
109          drone_nodes[i].lat = robotpose[i].lat
110          drone_nodes[i].lon = robotpose[i].lon
111          drone_nodes[i].alt = robotpose[i].alt
112
113      print("Sending location events to EMANE")
114      service = EventService(('224.1.2.8', 45703, 'control0')) # These values come
115      ↳ from EMANE config files
116      for drone in drone_nodes:
117          drone.location_event(service)
118
119  def communicate():
120      global shm_meta
121      global shm_comm
122      global robotcomm

```

```

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_emanet_comms",
128         ↪ create=False)
129         robotcomm = [RobotComms() for i in range(sys_meta.num_comms)]
130
131     print(sys_meta)
132     # BUG: Correlate list index with drone id (breaks if drone poses or drone IDs
133     ↪ are not in sequential order)
134
135     try:
136         for i in range(sys_meta.num_comms):
137             buf = shm_comm.buf[i*COMM_SIZE:(i+1)*COMM_SIZE]
138             robotcomm[i].unpack(buf)
139             # drone_nodes[0].inc_buffer(robotcomm[i].buff_size)
140             drone_nodes[i].buff = robotcomm[i].buff_size # NOTE:
141             robotcomm[i].sent = drone_nodes[i].do_broadcast()
142             buf = robotcomm[i].pack()
143             shm_comm.buf[i*COMM_SIZE:(i+1)*COMM_SIZE] = buf
144
145             # TODO: Check latency between all pairs and store to report to SySML
146
147     except:
148         pass
149
150 if __name__ == '__main__':
151     iternum = 0
152
153     init()
154     print(sys_meta)
155     while True:
156         wait_for_argos()
157         time.sleep(1)
158         update_drone()
159         communicate()

```



```

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 attempting 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
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():
7      '''
8          Corresponds to a single drone node
9          Holds its own ID, Location, and TxBufferSize
10
11      Attributes
12      -----
13      id - Drone node ID
14      lat - Node's latitude
15      lon - Node's longitude
16      alt - Node's altitude
17      buff - Number of bytes this node needs to still transmit
18      sent - Number of bytes this node sent this iteration
19      '''
20      id:  c_ushort = None
21      lat: c_double = None
22      lon: c_double = None
23      alt: c_double = None
24      buff: c_double = 0.0
25
26      def inc_buffer(self, size):
27          self.buff += size
28
29      def dec_buffer(self, size):
30          self.buff -= size
31
32
33      def location_event(self, service):
34          loc_event = LocationEvent()
35          loc_event.append(self.id, latitude=self.lat, longitude=self.lon,
36              ↪ altitude=self.alt, yaw=0, pitch=0, roll=0)
37          service.publish(0, loc_event)
38          return 0

```